What's a Linked List, Anyway? [Part 2]

Vaidehi Joshi · Follow
Published in basecs · 9 min read · Jan 23, 2017

This is the second installment in a two-part series on Linked Lists. If you haven't read Part 1 *of this series, I recommend checking that out first!*

Linked lists are super simple, but they seem to have this reputation of being fairly complicated. Their reputation, as they say, precedes them.

But the more that I've read about this specific data structure, the more I've realized that it's not actually *linked lists* that are confusing, but rather, it's the logic that goes into deciding when and whether or not to use them that's hard to wrap your head around.

If you've never worked with space time complexity (or if you're like me, and still struggle to understand it sometimes), an interview question like, *"Please tell me the how you'd implement X as a linked list, and what the possible*

drawbacks of that implementation are?" seems pretty...well, daunting, terrifying, near-impossible are all adjectives that come to mind.

So, let's deconstruct that scary reputation that linked lists seem to always have. Instead of being intimidated, we'll break them down and figure out what makes them useful, and when we might want to consider implementing them.

Hey, so, what even is Big O?

Most of us have probably heard the term "Big O Notation", even if we had no idea what it meant the first time that we heard someone use it. My personal experience with it has always been in the context of designing algorithms (or being asked to evaluate the efficiency of an algorithm). But Big O is really all over and omnipresent within computer science.

The reason for this is that computer science — and effectively, anything that we code — is all about efficiency and tradeoffs. Whether you're building software as a service, choosing a front end framework, or just trying to make your code DRY and more elegant, that's what all of us are striving towards: being efficient with our software, and choosing things that are important to what we're building, all while being aware of the tradeoffs that we'll ultimately have to make.

The same goes for Big O Notation, but on a much lower level. **Big O Notation** is a way of evaluating the performance of an algorithm.

> There are two major points to consider when thinking about how an algorithm performs: how much time it

> requires at runtime given how much time and memory it needs.

One way to think about Big O notation is a way to express the amount of time that a function, action, or algorithm takes to run based on how many elements we pass to that function.

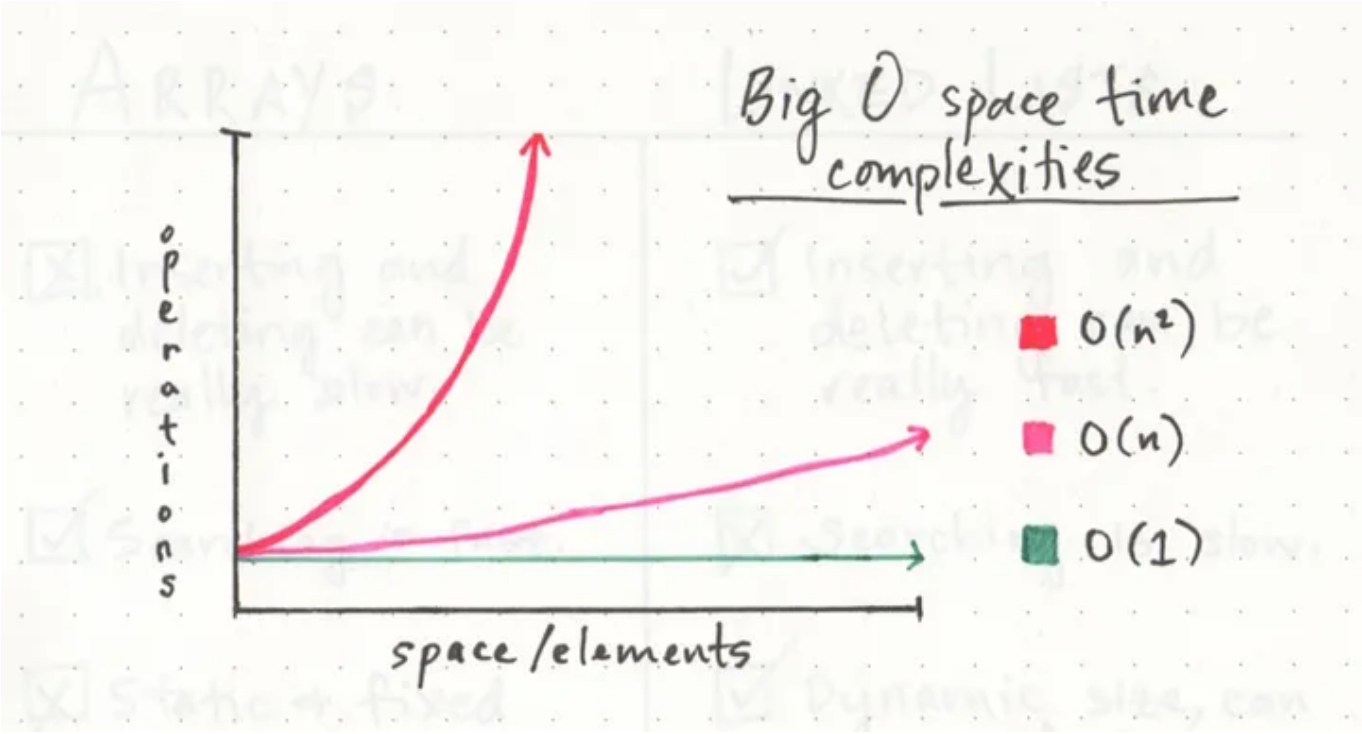
For example, if we have a list of the number 1–10, and we wanted to write an algorithm that multiplied each number by 10, we'd think about how much time that algorithm would take to multiply ten numbers. But what if instead of ten numbers, we had ten thousand? Or a million? Or tens of millions? That's exactly what Big O Notation takes into account: the speed and efficiency with which something functions when its input grows to be any (crazy big!) size.

I really love the way that Parker Phinney describes what Big O notation is used for in his awesome Interview Cake blog post. The way that he explains it, Big O Notation is all about the *way* an algorithm grows when it runs:

> *Some external factors affect the time it takes for a function to run: the speed of the processor, what else the computer is running, etc. So it's hard to make strong statements about the* exact runtime *of an algorithm. Instead we use big O notation to express* how quickly its runtime grows.

If you do a little bit of research on Big O Notation, you'll quickly find that there are a ton of different equations used to define the space and time complexity of an algorithm, and most of them involve an **O** (referred to as just "O" or sometimes as "order"), and a variable *n*, where *n* is the size of the input (think back to our our ten, thousands, or millions of numbers).

As far as linked lists go, however, the two types of Big O equations to remember are **O(1)** and **O(n)**.



Basic Big O Notation Equations

An **O(1) function** takes *constant* time, which is to say that it doesn't matter how many elements we have, or how huge our input is: it'll always take the same amount of time and memory to run our algorithm. An **O(n) function** is *linear*, which means that as our input grows (from ten numbers, to ten thousand, to ten million), the space and time that we need to run that algorithm grows linearly.

For a little contrast, we can also compare these two functions to something starkly different: an **O(n^2) function**, which clearly takes exponentially more time and memory the more elements that you have. It's pretty safe to say that we want to avoid O(n^2) algorithms, just from looking at that crazy red line!

Growing a linked list

We already know what linked lists are made of, and how their non-contiguous memory allocation makes them uniquely different from their seemingly more popular cousin, the array.

So how do they work, exactly? Well, just like with an array, we can add elements and remove elements from a linked list. But *unlike* arrays, we don't need to allocate memory in advance or copy and re-create our linked list, since we won't "run out of space" the way we might with a pre-allocated array.

Instead, all we really need to do is **rearrange our pointers**. We know that a linked list is made up a single node, and a node always contains some data and, most importantly, a pointer to the *next* node or null. So, all we need to do in order to *add* something to our linked list is figure out which pointer needs to point to where.

For simplicity's sake, we'll work with a singly linked list in these examples. We'll start with the simplest place we can insert an element into a linked list: at the very beginning. This is fairly easy to do, since we don't need to go through our entire list; instead we just start at the beginning.

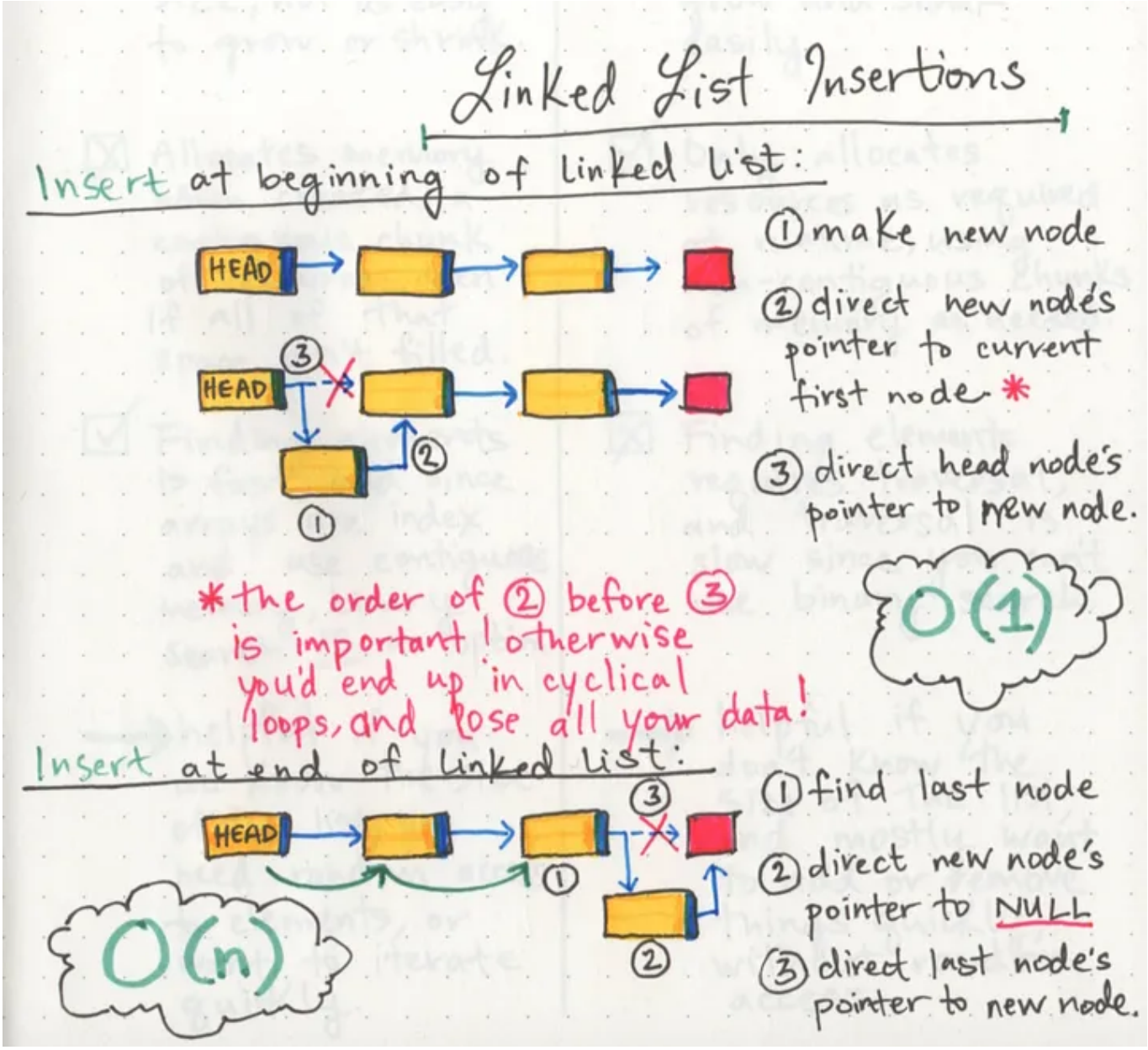
1. First, we find the head node of the linked list.
2. Next, we'll make our new node, and set its pointer to the *current* first node of the list.
3. Lastly, we rearrange our head node's pointer to point at our new node.

That's it, we're done! Well, almost. This looks easy, and it is — just as long as we do those three steps in the right order. If we accidentally end up doing step 3 before step 2, we end up in a bit of a mess. The reason being that if we

point our head node to the new node *before* connecting the new, still-to-be-inserted, node to the rest of the list that comes afterwards, we'd end up in a cyclical structure, with only two nodes, and we'd effectively lose *all* of our other data in the list. Talk about a bad day!

Still, that process isn't too hard to implement (and hopefully shouldn't be too difficult to code) once you can remember those three steps.

Inserting an element at the beginning of a linked list is particularly nice and efficient because **it takes the same amount of time, no matter how long our list is**, which is to say it has a space time complexity that is *constant*, or **O(1)**.



Inserting elements at the beginning and end of a linked list

But inserting an element at the end of a linked list is a different story. The interesting thing here is that the steps you take to actually *do* the inserting are the exact same:

1. Find the node we want to change the pointer of (in this case, the last node)
2. Create the new node we want to insert and set its pointer (in this case, to null)

3. Direct the preceding node's pointer to our new node

However, there's an added complexity here: we're not adding something to the beginning of the list, at the head node. Nope — instead, we're adding something after the *last* node. So we will need to find the last node. Which means that we'll need to traverse through the *entire linked list* to find it. And we know that linked lists are distributed, non-contiguous in memory, which means that each node will live at a totally different "address" in memory! So traversing is going to take time — it's going to take more time with the more nodes we have.

Hopefully, we can start to see what kind of space time complexity this type of inserting will leave us with: a linear **O(n)**. If we had a linked list of 100 nodes, that might not actually take that long. Even a 1000 might be pretty fast. But imagine if we wanted to add an element to the end of a linked list with a billion items! This insert algorithm would take *as much time as the number of elements in our list*, which, depending on our list, could be a very bad day for us.

To list or not to list?

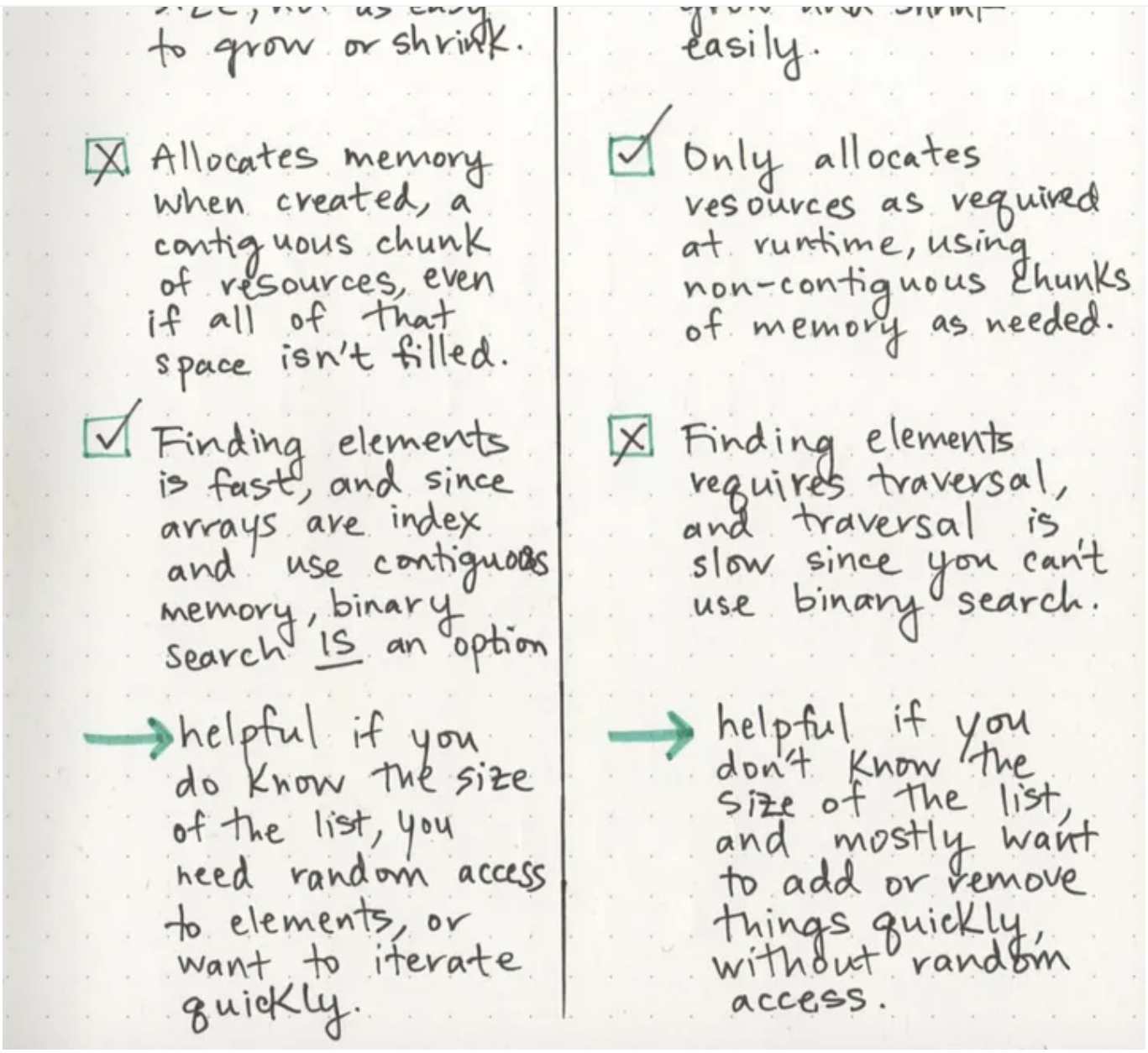
No human is perfect, and neither is a linked list. Here's the thing: sometimes, a linked list can be really awesome — for example, when you want to insert or remove something at the beginning of the list. But, as we've learned, they can sometimes be…less than ideal (imagine having a million nodes and just wanting to delete the last one!)

Even if you don't have to work with them every day, it's useful to know just enough about data structures like linked lists so that you can both recognize them when you see them, and know when to reach for them when the time is right.

A good rule of thumb for remember the characteristics of linked lists is this:

> a linked list is usually efficient when it comes to adding and removing most elements, but can be very slow to search and find a single element.

If you ever find yourself having to do something that requires a lot of traversal, iteration, or quick index-level access, a linked list could be your worst enemy. In those situations, an **array** might be a better solution, since you can find things quickly (a single chunk of allocated memory), and you can use an index to quickly retrieve a random element in the middle or end of the list without having to take the time to traverse through the whole entire thing.



However, if you find yourself wanting to add a bunch of elements to a list and aren't worried about finding elements again later, or if you know that you won't need to traverse through the entirety of the list, a **linked list** could be your new best friend.

For the longest time, I never knew what linked lists were. And when I finally did learn about them, I didn't know what on earth they would ever be used for. Now that I realize what they do well, I can see both their benefits and drawbacks. And if the time ever comes, I'll know when to reach for them to help me out.

Hopefully, you're with me on this and feel the same way about linked lists, too!

Resources

If you think that space time complexity and linked lists are dope, you might think that the below links are, also, quite dope. And perhaps even helpful! Some of these links were recommended in Part 1 of this series, but you'll find that they apply to Part 2, as well.

Happy listing!

1. A Beginner's Guide To Big O Notation, Rob Bell
2. Big O Cheat Sheet, Eric Rowell
3. Ruby's Missing Data Structure, Pat Shaughnessy
4. When to use a linked list over an array/array list?, StackOverflow

5. Data Structures: Arrays vs Linked Lists, mycodeschool
6. Static Data Structures vs. Dynamic Data Structures, Ayoma Gayan Wijethunga

Programming | Data Structures | Computer Science | Tech | Software Development

Written by Vaidehi Joshi

29K Followers · Editor for basecs

Writing words, writing code. Sometimes doing both at once.

Follow

More from Vaidehi Joshi and basecs