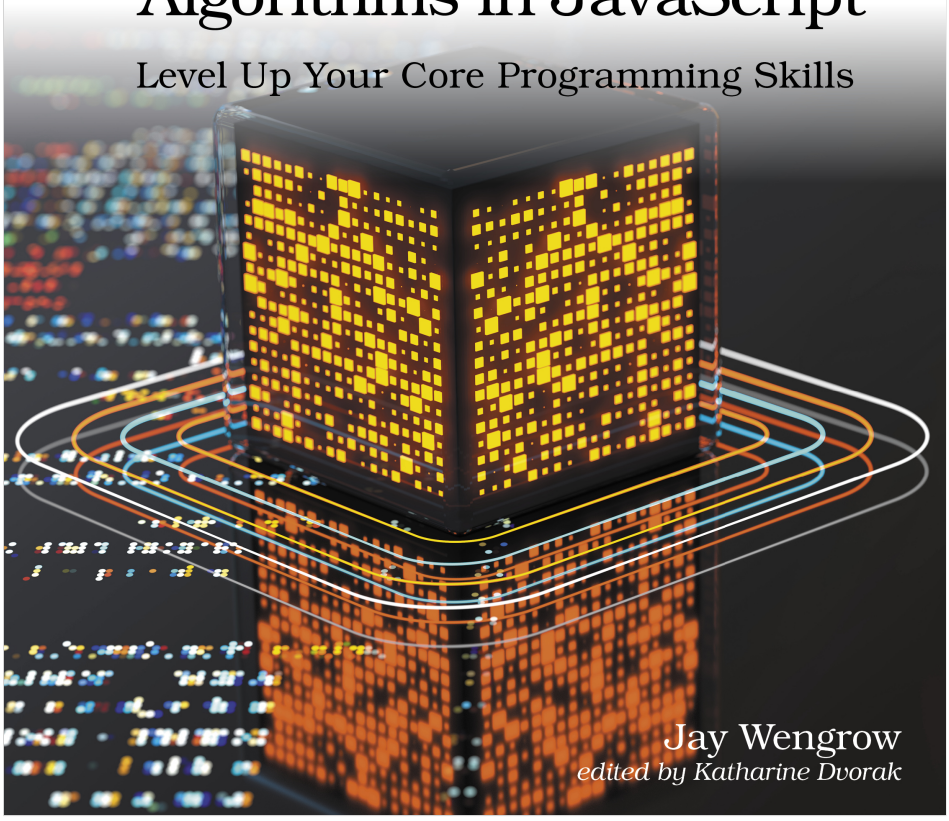
The Pragmatic Programmers

Volume 1

A Common-Sense Guide to

Data Structures and Algorithms in JavaScript

Level Up Your Core Programming Skills



Jay Wengrow

edited by Katharine Dvorak

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 14

Node-Based Data Structures

For the next several chapters, we're going to explore a variety of data structures that all build upon a single concept—the *node*. As you'll see shortly, nodes are pieces of data that may be dispersed throughout the computer's memory. Node-based data structures offer new ways to organize and access data that provide a number of major performance advantages.

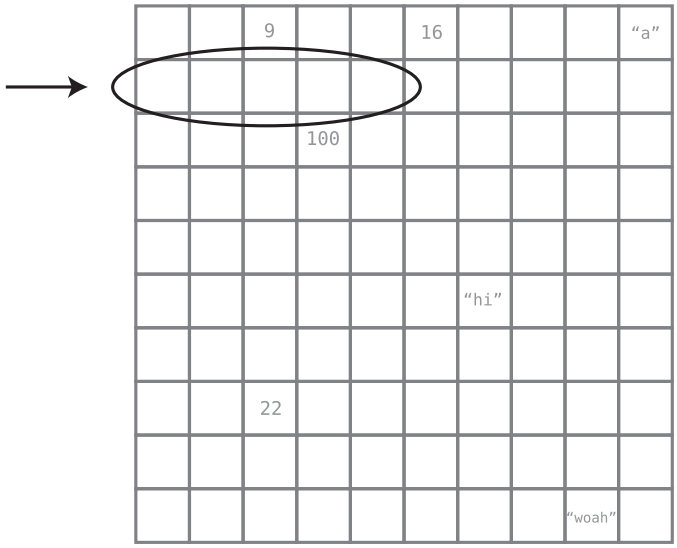
In this chapter, we'll explore the linked list, which is the simplest node-based data structure and the foundation for future chapters. You'll discover that linked lists seem almost identical to arrays but come with their own set of trade-offs in efficiency that can give us a performance boost for certain situations.

Linked Lists

Like an array, a *linked list* is a data structure that represents a list of items. While on the surface arrays and linked lists look and act quite similarly, under the hood there are big differences.

As mentioned in Chapter 1, Why Data Structures Matter, on page ?, memory inside a computer can be visualized as a giant set of cells in which bits of data are stored. You learned that when creating an array, your code finds a contiguous group of empty cells in memory and designates them to store data for your application, as shown on page 4.

You also saw that the computer has the ability to access any memory address in one step and can use that power to also immediately access any index within the array. If you wrote code that said, "Look up the value at index 4," your computer could locate that cell in a single step. Again, this is because your program knows which memory address the array starts at—say, memory address 1000—and therefore knows that if it wants to look up index 4, it should simply jump to memory address 1004.

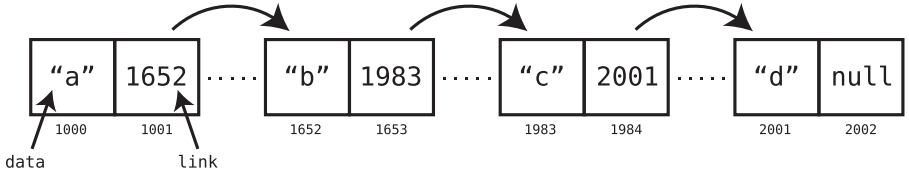


Linked lists, on the other hand, work differently. Instead of being a contiguous block of memory, the data from linked lists can be scattered across different cells throughout the computer's memory.

Connected data that are dispersed throughout memory are known as *nodes*. In a linked list, each node represents one item in the list. The big question, then, is this: if the nodes are not next to each other in memory, how does the computer know which nodes are part of the same linked list?

This is the key to the linked list: each node also comes with a little extra information—namely, the memory address of the *next* node in the list.

This extra piece of data—this pointer to the next node's memory address—is known as a *link*. Here's a visual depiction of a linked list:



In this example, we have a linked list that contains four pieces of data: `"a"`, `"b"`, `"c"`, and `"d"`. However, it uses *eight* cells of memory to store this data, because each node consists of two memory cells. The first cell holds the actual data, while the second cell serves as a link that indicates where in

memory the next node begins. The final node's link contains `null` since the linked list ends there.

A linked list's first node is also referred to as its *head*, and its last node its *tail*. We may use the terms *head* and *first node* interchangeably.

If the computer knows at which memory address the linked list begins, it has all it needs to begin working with the list! Since each node contains a link to the next node, all the computer needs to do is follow each link to string together the entire list.

The fact that a linked list's data can be spread throughout the computer's memory is a potential advantage it has over the array. An array, by contrast, needs to find an entire block of contiguous cells to store its data, which can get increasingly difficult as the array size grows. These details are managed by your programming language under the hood, so you may not have to worry about them. However, you'll see shortly that there are more tangible differences between linked lists and arrays that we can sink our teeth into.

Implementing a Linked List

Some programming languages, such as Java, come with linked lists built into the language. Many languages don't, but it's fairly simple to implement them on our own.

Let's create our own linked list using JavaScript. We'll use two classes to implement this: `Node` and `LinkedList`. Let's create the `Node` class first:

```
class Node {
  constructor(data) {
    this.data = data;
    this.nextNode = null;
  }
}

export default Node;
```

The `Node` class has two attributes: `data` contains the node's primary value (for example, the string `"a"`), while `nextNode` contains the link to the next node in the list. We can use this class as follows:

```
const node1 = new Node('once');
const node2 = new Node('upon');
const node3 = new Node('a');
const node4 = new Node('time');

node1.nextNode = node2;
node2.nextNode = node3;
node3.nextNode = node4;
```

With this code, we've created a list of four nodes that serve as a list containing the strings 'once', 'upon', 'a', and 'time'.

Note that in our implementation, the nextNode refers to another Node *instance* rather than an actual memory address number. The effect, however, is the same—the nodes are likely dispersed throughout the computer's memory, and yet we can use the nodes' links to string the list together.

Going forward, then, we're simply going to discuss each link as pointing to another node rather than to a specific memory address. Accordingly, we're going to use simplified diagrams to depict linked lists, such as this one:

Each node in this diagram consists of two cells. The first cell contains the node's data, and the second cell points to the next node.

This reflects our implementation of the Node class. In it, the data method returns the node's data, while the nextNode method returns the next node in the list. *In this context, the nextNode method serves as the node's link.*

While we've been able to create this linked list with the Node class alone, we still need an easy way to tell our program where the linked list begins. To do this, we'll create a LinkedList class in addition to our previous Node class. Here's the LinkedList class in its basic form:

```
import Node from './node.js';

class LinkedList {
  constructor(firstNode=null) {
    this.firstNode = firstNode;
  }

}
```

Note that we import node at the beginning of our code, since we placed the Node class in a separate file from this LinkedList class.

At this point, all a LinkedList instance does is keep track of the first node of the list.

Previously we created a chain of nodes containing node1, node2, node3, and node4. We can now use our LinkedList class to reference this list by writing the following code:

```
const list = new LinkedList(node1);
```

This list variable now acts as a handle on the linked list, as it's an instance of LinkedList that has access to the list's first node.

An important point emerges: *when dealing with a linked list, we have immediate access only to its head.* This is going to have serious ramifications, as we'll see shortly.

At first glance, though, linked lists and arrays are similar—they're both just lists of stuff. When we dig into the analysis, though, we'll see some dramatic differences in these two data structures' performances! Let's jump into the four classic operations: reading, searching, insertion, and deletion.