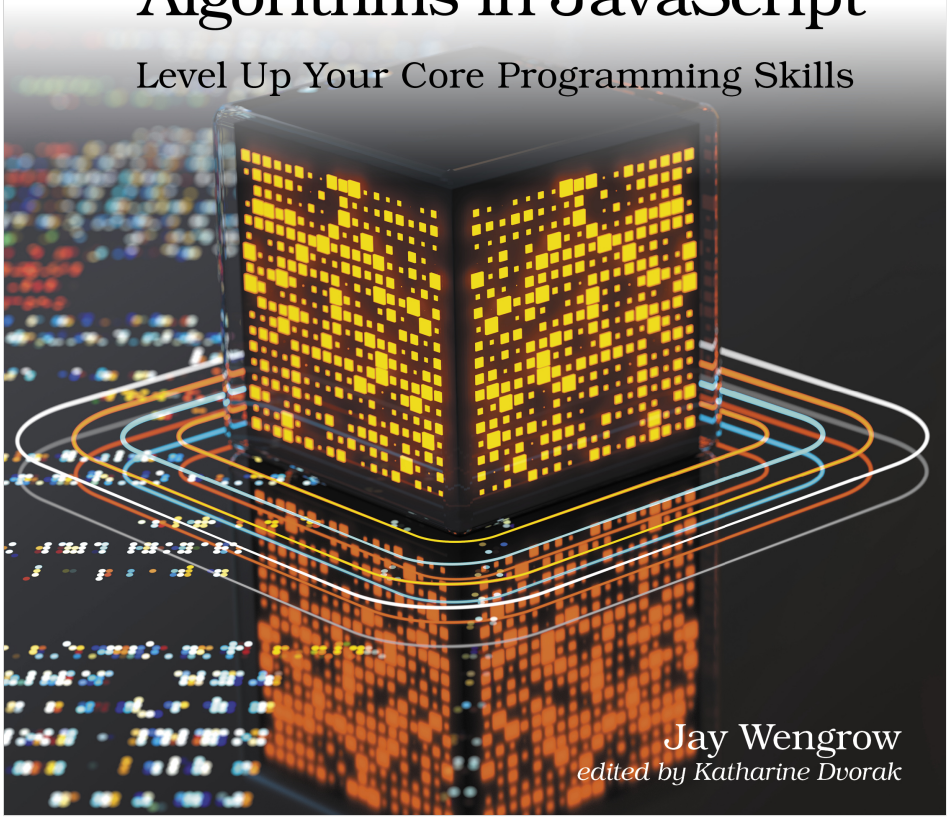
The Pragmatic Programmers

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.

Copyright © The Pragmatic Programmers, LLC.

CHAPTER 8

Blazing Fast Lookup with Hash Tables

Imagine you're writing a program that allows customers to order fast food from a restaurant, and you're implementing a menu of foods with their respective prices. You could, technically, use an array:

```
menu = [
    ["french fries", 0.75],
    ["hamburger", 2.5],
    ["hot dog", 1.5],
    ["soda", 0.6]
]
```

This array contains several subarrays, and each subarray contains two elements. The first element is a string representing the food on the menu, and the second element represents the price of that food.

As you learned in Chapter 2, Why Algorithms Matter, on page ?, if this array were unordered, searching for the price of a given food would take O(N) steps since the computer would have to perform a linear search. If it's an *ordered* array, the computer could do a binary search, which would take O(log N).

While O(log N) isn't bad, we can do better. In fact, we can do *much* better. By the end of this chapter, you'll learn how to use a special data structure called a *hash table*, which can be used to look up data in just O(1) time. By knowing how hash tables work under the hood and the right places to use them, you can leverage their tremendous lookup speeds in many situations.

Hash Tables

Most programming languages include a data structure called a *hash table*, and it has an amazing superpower: fast reading. Note that hash tables are called by different names in various programming languages. In JavaScript they're called *objects*, and other languages call them hashes, maps, hash

• Click HERE to purchase this book now. discuss

maps, dictionaries, or associative arrays. We'll refer to them as hash tables since that's a common universal way to refer to this data structure.

Here's an example of the menu as implemented with a hash table:

```
const menu = { "french fries": 0.75, "hamburger": 2.5,
"hot dog": 1.5, "soda": 0.6 };
```

A hash table is a list of paired values. The first item in each pair is called the *key*, and the second item is called the *value*. In a hash table, the key and value have some significant association with one another. In this example, the string "french fries" is the key, and 0.75 is the value. They are paired together to indicate that french fries cost 75 cents.

In JavaScript, you can look up a key's value using this syntax:

```
menu["french fries"];
```

This would return the value 0.75. If we tried to look up a value that doesn't exist with the hash table, we'd get back undefined.

Looking up a value in a hash table has an efficiency of O(1) on average, as it usually takes *just one step*. Let's see why.

Hashing with Hash Functions

Do you remember those secret codes you used as a kid to create and decipher messages? For example, here's a simple way to map letters to numbers:

A = 1
B = 2
C = 3
D = 4
E = 5

and so on. According to this code,

ACE converts to 135,
CAB converts to 312,
DAB converts to 412,

and

BAD converts to 214.

This process of taking characters and converting them to numbers is known as *hashing*. And the code that is used to convert those letters into particular numbers is called a *hash function*.

Many other hash functions exist besides this one. Another example of a hash function may be to take each letter's corresponding number and return the *sum* of all the numbers. If we did that, BAD would become the number 7, following a two-step process:

Step 1: First, BAD converts to 214.

Step 2: We then take each of these digits and get their sum:

2 + 1 + 4 = 7

Another example of a hash function may be to return the *product* of all the letters' corresponding numbers. This would convert the word BAD into the number 8:

Step 1: First, BAD converts to 214.

Step 2: We then take the product of these digits:

2 * 1 * 4 = 8

In our examples for the remainder of this chapter, we're going to stick with this last version of the hash function. Real-world hash functions are more complex than this, but this multiplication hash function will keep our examples clear and simple.

The truth is that a hash function needs to meet only one criterion to be valid: a hash function must convert the same string to the *same number* every single time it's applied. If the hash function can return inconsistent results for a given string, it's not valid.

Examples of invalid hash functions include functions that use random numbers or the current time as part of their calculation. With these functions, BAD might convert to 12 one time and 106 another time.

With our multiplication hash function, however, BAD will *always* convert to 8. That's because B is always 2, A is always 1, and D is always 4. And 2 * 1 * 4 is *always* 8. There's no way around this.

Note that with this hash function, DAB will *also* convert into 8 just as BAD will. This will cause some issues that I'll address later.

• Click HERE to purchase this book now. discuss

Armed with the concept of hash functions, we can now understand how a hash table works.

• Click HERE to purchase this book now. discuss