HASH MAPS: CONCEPTUAL

Hash Map Methodology

In the case of a map between two things, we don't really care about the exact *sequence* of the data. We only care that a given input, when fed into the map, gives the accurate output. Developing a data structure that performs this is tricky because computers care much more about values than relationships. A computer doesn't really care to memorize the astrological signs of all of our friends, so we need to trick the computer into caring.

We perform this trick using a structure that our computer is already familiar with, an array. An array uses indices to keep track of values in memory, so we'll need a way of turning each key in our map to an index in our array.

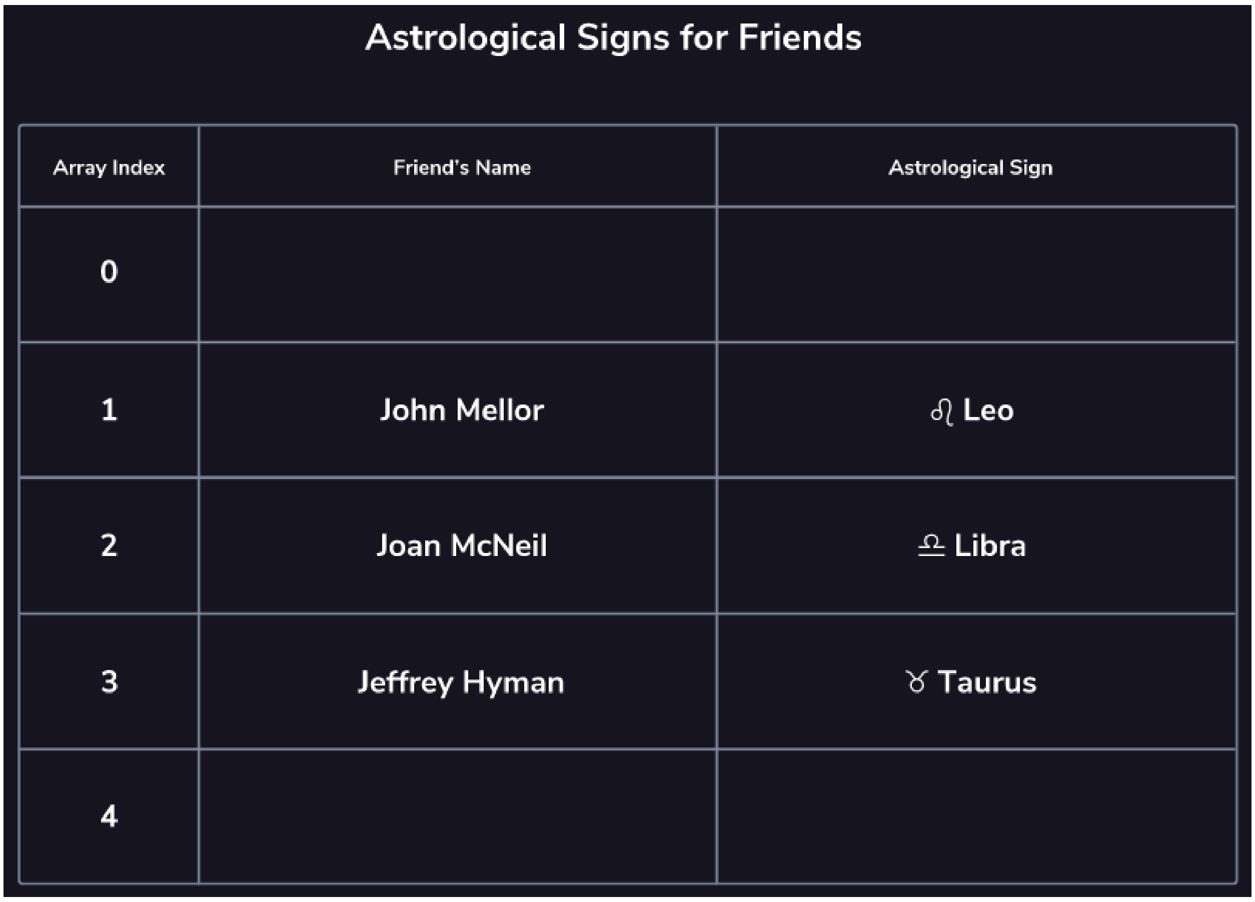
Imagine we want our computer to remember that our good friend Joan McNeil is a Libra. We take her name, and we turn that name into a number. Let's say that the number we correspond with the name "Joan McNeil" is 17. We find the 17th index of the array we're using to store our map and save the value (Libra) there.

How did we get 17, though? We use a special function that turns data like the string "Joan McNeil" into a number. This function is called a *hashing function*, or a hash function. Hashing functions are useful in many domains, but for our data structure the most important aspect is that a hashing function returns an array index as output.

Instructions

What are some hashing functions we can build with the tools we have so far? What if we add up the number of vowels in the key? What if we assign a value to every letter in the alphabet (say A=1, B=2, etc.,) and add up all of those values? What if the string "Codecademy" returns 1 and all other strings return 0? All of these are hashing functions (some admittedly much more useful than others).

Does the key to a hash table need to be a string? Can you define a hash function that takes a different kind of input?



Astrological Signs for Friends

Array Index	Friend's Name	Astrological Sign
0		
1	John Mellor	♌ Leo
2	Joan McNeil	♎ Libra
3	Jeffrey Hyman	♉ Taurus
4		

Hash Functions

A hash function takes a string (or some other type of data) as input and returns an array index as output. In order for it to return an array index, our hash map implementation needs to know the size of our array. If the array we are saving values into only has 4 slots, our hash map's hashing method should not return an index bigger than that.

In order for our hash map implementation to guarantee that it returns an index that fits into the underlying array, the hash function will first compute a value using some scoring metric: this is the hash value, hash code, or just the *hash*. Our hash map implementation then takes that hash value mod the size of the array. This guarantees that the value returned by the hash function can be used as an index into the array we're using.

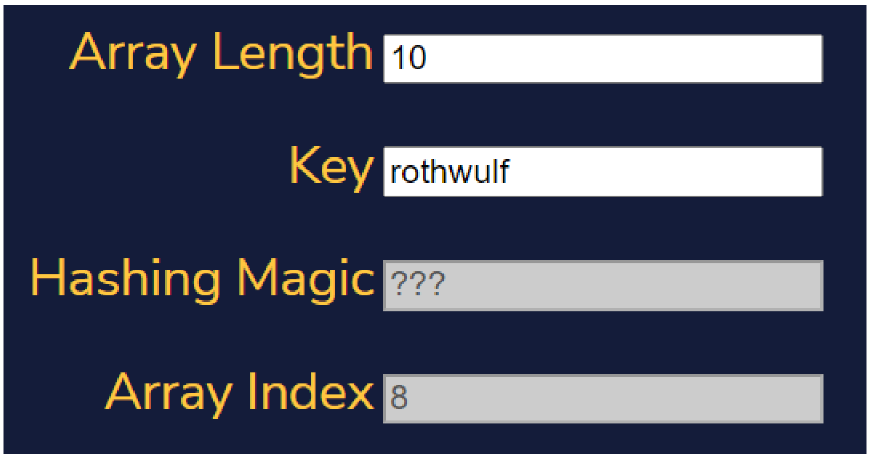
It is actually a defining feature of all hash functions that they greatly reduce any possible inputs (any string you can imagine) into a much smaller range of potential outputs (an integer smaller than the size of our array). For this reason, hash functions are also known as *compression functions*.

Much like an image that has been shrunk to a lower resolution, the output of a hash function contains less data than the input. Because of this, hashing is not a reversible process. With just a hash value it is impossible to know for sure the key that was plugged into the hashing function.

Instructions

Why is it necessary for a hash function to not be reversible? Does this “only works in one direction” criterion sound familiar?

Change the key and array size in the application. See how different keys give different indices, but sometimes different keys give the same index. Notice that the array index is always smaller than the length of the array.



Array Length 10

Key rothwulf

Hashing Magic ???

Array Index 8

How to Write a Hash Function

You might be thinking at this point that we’ve glossed over a very important aspect of a hash table here. We’ve mentioned that a hash function is necessary, and described some features of what a hash function does, but never really given an implementation of a hash function that does not feel like a toy example.

Part of this is because a hash function needs to be simple by design. Performing complex mathematical calculations that our hash table needs to compute every time it wants to assign or retrieve a value for a key will significantly damage a hash table’s performance for two things that it should be able to do quickly.

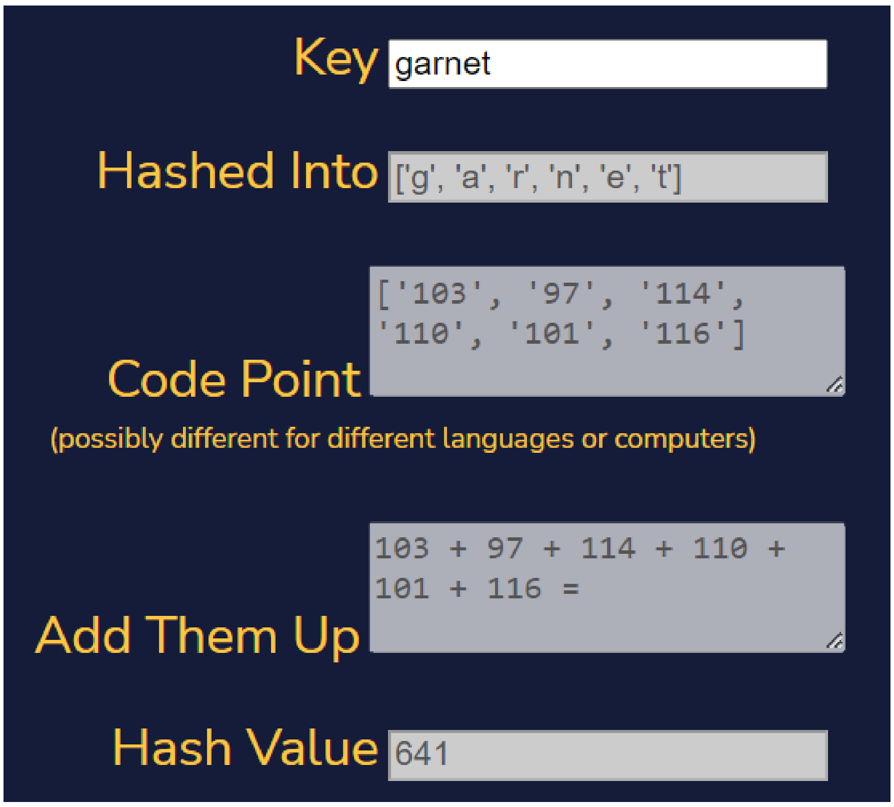
Hash functions also need to be able to take whatever types of data we want to use as a key. We only discussed strings, a very common use case, but it’s possible to use numbers as hash table keys as well.

A very common hash function for integers, for example, is to perform the modular operation on it to make sure it’s less than the size of the underlying array. If the integer is already small enough to be an index into the array, there's nothing to be done.

Many hash functions implementations for strings take advantage of the fact that strings are represented internally as numerical data. Frequently a hash function will perform a shift of the data bitwise, which is computationally simple for a computer to do but also can predictably assign numbers to strings.

Instructions

In the application change the `Key` and watch how a hashing function might create a hash value for that key.



Key: garnet

Hashed Into: ['g', 'a', 'r', 'n', 'e', 't']

Code Point: ['103', '97', '114', '110', '101', '116']

(possibly different for different languages or computers)

Add Them Up: 103 + 97 + 114 + 110 + 101 + 116 =

Hash Value: 641

Basic Hash Maps

Now that we have all of the main ingredients for a hash map, let's put them all together. First, we need some sort of associated data that we're hoping to preserve. Second, we need an array of a fixed size to insert our data into. Lastly, we need a hash function that translates the keys of our array into indexes into the array. The storage location at the index given by a hash is called the *hash bucket*.

Let's use the following example for our hash map:

Key: Album Name	Value: Release Year
The Low End Theory	1991
Midnight Marauders	1993
Beats, Rhymes and Life	1996
The Love Movement	1998

Our map here relates to several A Tribe Called Quest studio albums with the year they were produced in. We’ll need an array of at least size 4 to contain all of these elements. And a way to turn each album name into an index into that array.

For each album name, find that album’s hash by performing the following calculation:

```
hash_value = ((# of lowercase 'a's in album name) + (# of number of lowercase 'e's in album name))
```

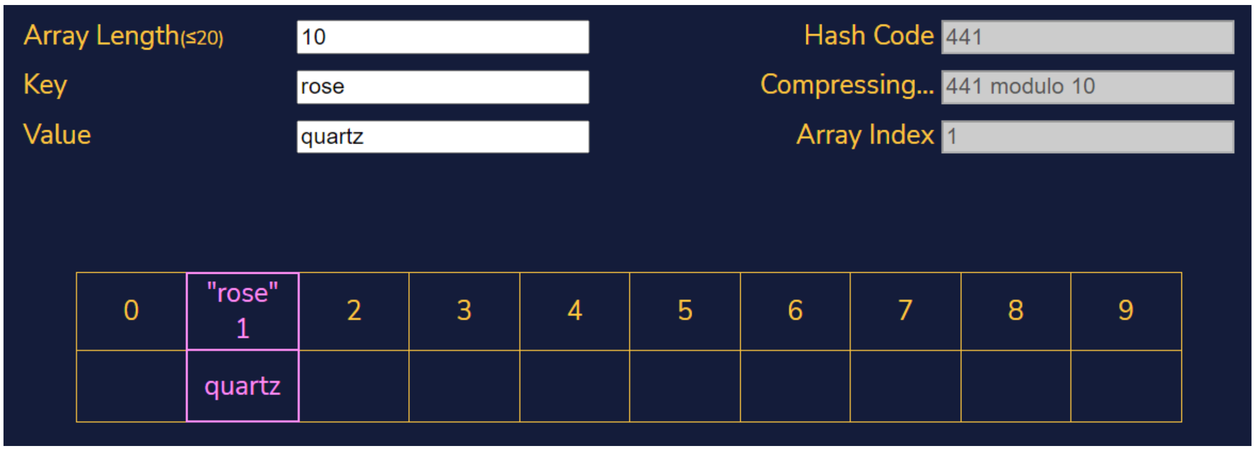
And then take that hash and calculate an array index by performing `hash_value mod 4`. Following these steps we get the following schema:

Album Name	Hash	Hash mod 4	Release Year
The Low End Theory	2	2	1991
Midnight Marauders	3	3	1993
Beats, Rhymes and Life	5	1	1996
The Love Movement	4	0	1998

First, the key is translated into the hash using our hashing function. Then, our hash map performs modulo arithmetic to turn the hash into an array index.

Instructions

Where would you save the value for a given key? It depends on the length of the array and the key itself. Update the values for those two to see how different keys and array lengths change the place a key is saved.



Array Length(≤20): 10
Key: rose
Value: quartz
Hash Code: 441
Compressing...: 441 modulo 10
Array Index: 1

0	"rose" 1	2	3	4	5	6	7	8	9
	quartz								

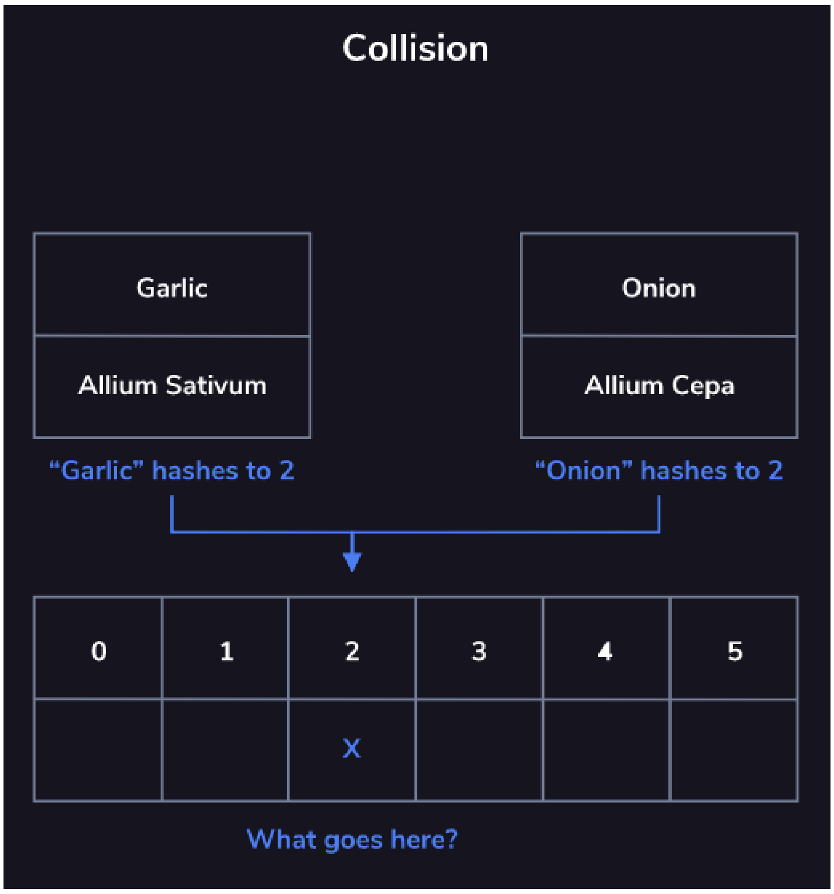
Collisions

Remember hash functions are designed to compress data from a large number of possible keys to a much smaller range. Because of this compression, it's likely that our hash function might produce the same hash for two different keys. This is known as a *hash collision*. There are several strategies for resolving hash collisions.

The first strategy we're going to learn about is called *separate chaining*. The separate chaining strategy avoids collisions by updating the underlying data structure. Instead of an array of values that are mapped to by hashes, it could be an array of linked lists!

Instructions

When is it most likely for a collision to happen? Can a collision happen when the array is empty?



Separate Chaining

A hash map with a linked list separate chaining strategy follows a similar flow to the hash maps that have been described so far. The user wants to assign a value to a key in the map. The hash map takes the key and transforms it into a hash code. The hash code is then converted into an index to an array using the modulus operation. If the value of the array at the hash function's returned index is empty, a new linked list is created with the value as the first element of the linked list. If a linked list already exists at the address, append the value to the linked list given.

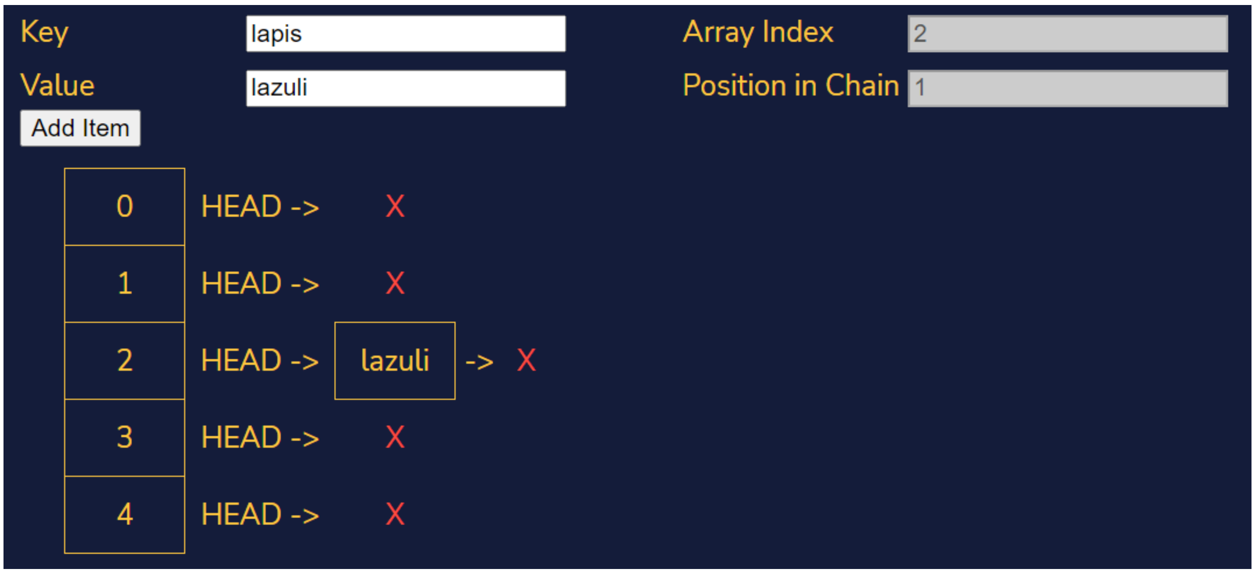
This is effective for hash functions that are particularly good at giving unique indices, so the linked lists never get very long. But in the worst-case scenario, where the hash function gives all keys the same index, lookup performance is only as good as it would be on a linked list. Hash maps are frequently employed because looking up a value (for a given key) is quick. Looking up a value in a linked list is much slower than a perfect, collision-free hash map of the same size. A hash map that uses separate chaining with linked lists but experiences frequent collisions loses one of its most essential features.

Instructions

In the application, we've implemented an array with separate chaining. Notice when two keys hash to the same value they get added to the chained linked list.

Separate chaining doesn't require linked lists to be the underlying data structure for each array index. What would a separate chaining algorithm with a different data structure look like? What would the benefits of using a tree at each index? What would be the drawbacks?

What if at every index we had another hash table?



Saving Keys

A hash collision resolution strategy like separate chaining involves assigning two keys with the same hash to different parts of the underlying data structure. How do we know which values relate back to which keys? If the linked list at the array index given by the hash has multiple elements, they would be indistinguishable to someone with just the key.

If we save both the key and the value, then we will be able to check against the saved key when we're accessing data in a hash map. By saving the key with the value, we can avoid situations in which two keys have the same hash code where we might not be able to distinguish which value goes with a given key.

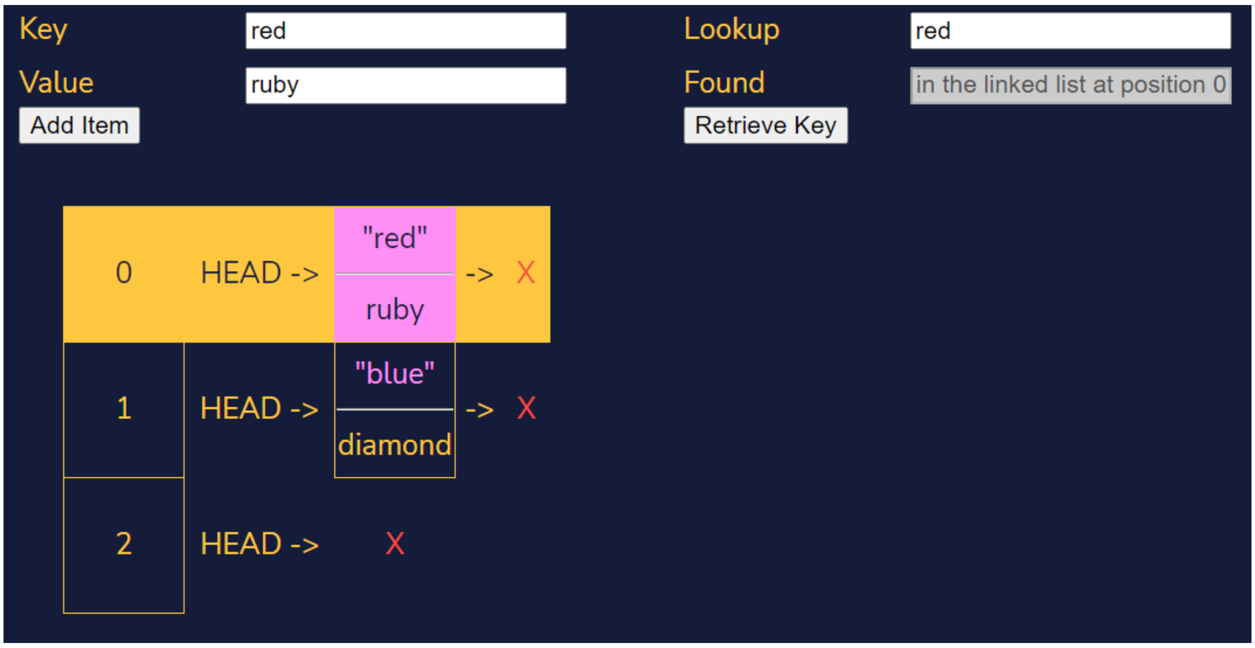
Now, when we go to read or write a value for a key we do the following: calculate the hash for the key, find the appropriate index for that hash, and begin iterating through our linked list. For each element, if the saved key is the same as our key, return the value. Otherwise, continue iterating through the list comparing the keys saved in that list with our key.

Instructions

Now that we save our keys, we can have it so that assigning to the same key overwrites the original key-value pair.

We can also retrieve values consistently.

Add a few keys and values to the hash map and then look up a key.



Open Addressing: Linear Probing

Another popular hash collision strategy is called *open addressing*. In open addressing we stick to the array as our underlying data structure, but we continue looking for a new index to save our data if the first result of our hash function has a different key's data.

A common open method of open addressing is called *probing*. Probing means continuing to find new array indices in a fixed sequence until an empty index is found.

Suppose we want to associate famous horses with their owners. We want our first key, "Bucephalus", to store our first value, "Alexander the Great". Our hash function returns an array index 3 and so we save "Alexander the Great", along with our key "Bucephalus", into the array at index 3.

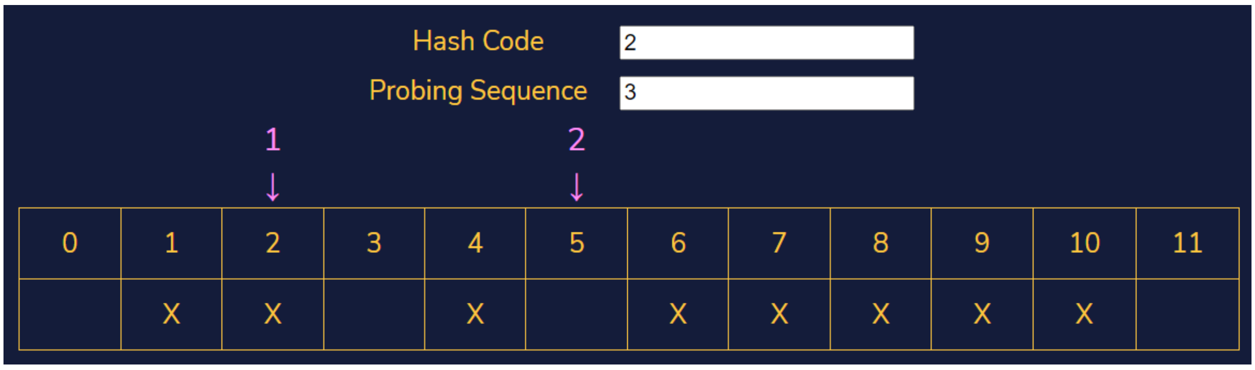
After that, we want to store “Seabiscuit”s owner “Charles Howard”. Unfortunately “Seabiscuit” also has a hash value of 3. Our probing method adds one to the hash value and tells us to continue looking at index 4. Since index 4 is open we store “Charles Howard” into the array at index 4. Because “Seabiscuit” has a hash of 3 but “Charles Howard” is located at index 4, we must also save “Seabiscuit” into the array at that index.

When we attempt to look up “Seabiscuit” in our Horse Owner’s Hash Map, we first check the array at index 3. Upon noticing that our key (Seabiscuit) is different from the key sitting in index 3 (Bucephalus), we realize that this can’t be the value we were looking for at all. Only by continuing to the next index do we check the key and notice that at index 4 our key matches the key saved into the index 4 bucket. Realizing that index 4 has the key “Seabiscuit” means we can retrieve the information at that location, Seabiscuit’s owner’s name: Charles Howard.

Instructions

If we add a third key, “Secretariat”, which also hashes to 3, where would the value be saved? What if we want to add a fourth key, “Epona” which returns from our hash function with an index of 5? When do you think we would use probing over other strategies?

In the workspace we’ve given an example of a hash map with a linear probing implementation. Some of the underlying array values of the hash map are already taken. Investigate where a key with a given hash code would be saved for different probing sequences.


Hash Code: 2

Probing Sequence: 3

		1 ↓			2 ↓						
0	1	2	3	4	5	6	7	8	9	10	11
	X	X		X		X	X	X	X	X	

Other Open Addressing Techniques

There are more sophisticated ways to find the next address after a hash collision, although anything too calculation-intensive would negatively affect a hash table’s performance. Linear probing systems, for instance, could jump by five steps instead of one step.

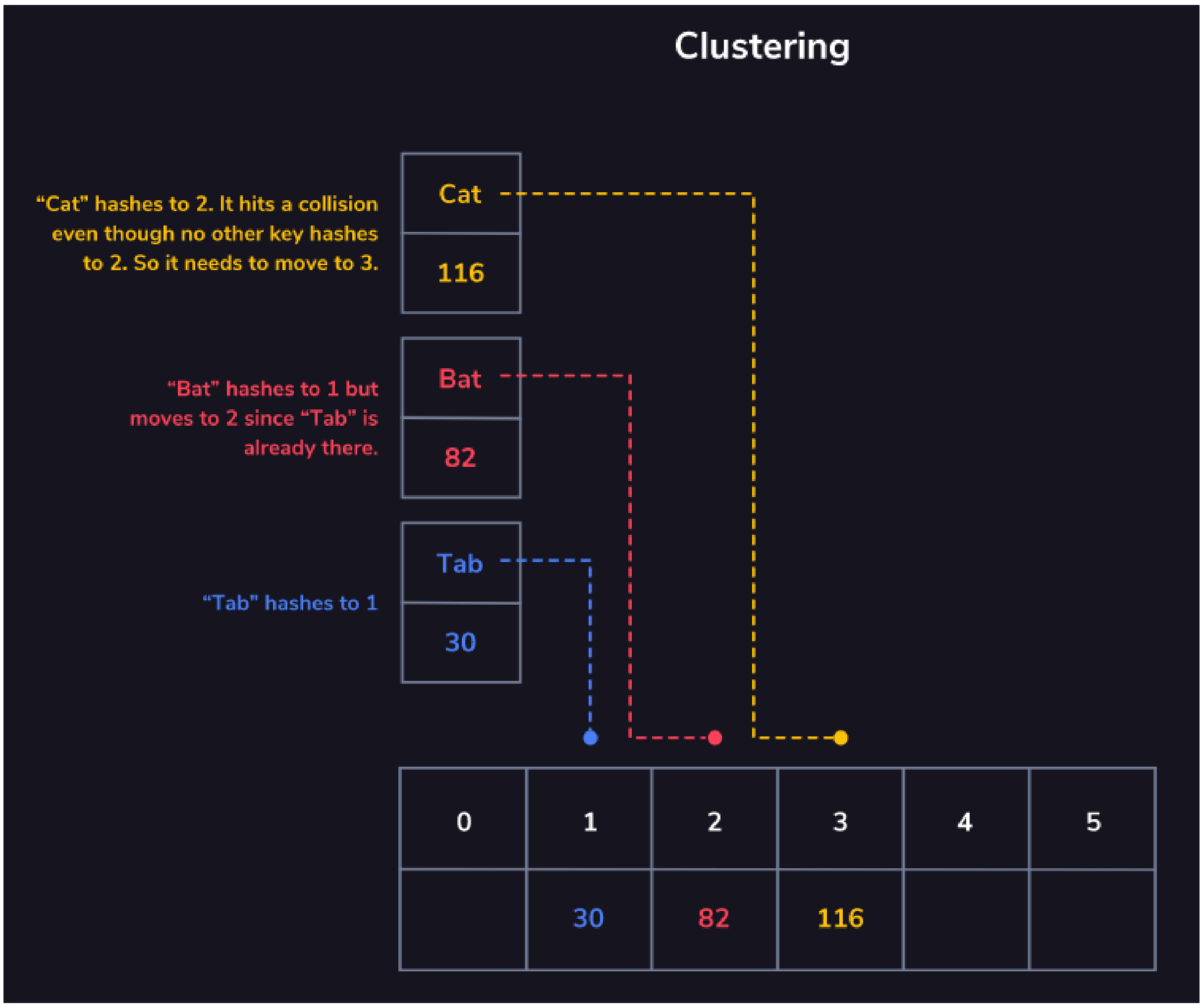
In a quadratic probing open addressing system, we add increasingly large numbers to the hash code. At the first collision we just add 1, but if the hash collides there too we add 4 ,and the third time we add 9. Having a probe sequence change over time like this avoids clustering.

Clustering is what happens when a single hash collision causes additional hash collisions. Imagine a hash collision triggers a linear probing sequence to assigns a value to the next hash bucket over. Any key that would hash to this “next bucket” will now collide with a key that, in a sense, doesn’t belong to that bucket anyway.

As a result the new key needs to be assigned to the next, next bucket over. This propagates the problem because now there are two hash buckets taken up by key-value pairs that were assigned as a result of a hash collision, displacing further pairs of information we might want to save to the table.

Instructions

What other ways can we iterate through addresses in our underlying way? Maybe we could write a function that takes the key and returns a number of steps to jump to get to the next index. What would you call a function that transforms a string into a number in this way?


Clustering
"Cat" hashes to 2. It hits a collision even though no other key hashes to 2. So it needs to move to 3.
Cat
116
"Bat" hashes to 1 but moves to 2 since "Tab" is already there.
Bat
82
"Tab" hashes to 1
Tab
30
0
1
2
3
4
5
30
82
116

Review

We've learned together what a hash map is and how to create one. Let's go over the concepts presented in this lesson.

A hash map is:

- Built on top of an array using a special indexing system.
- A key-value storage with fast assignments and lookup.
- A table that represents a map from a set of keys to a set of values.

Hash maps accomplish all this by using a hash function, which turns a key into an index into the underlying array.

A hash collision is when a hash function returns the same index for two different keys.

There are different hash collision strategies. Two important ones are separate chaining, where each array index points to a different data structure, and open addressing, where a collision triggers a probing sequence to find where to store the value for a given key.

Instructions

What makes good data to save into a hash map? Can you think of instances where a hash map would help you solve a problem faster?

review.txt

Hash map: A key-value store that uses an array and a hashing function to save and retrieve values.

Key: The identifier given to a value for later retrieval.

Hash function: A function that takes some input and returns a number.

Compression function: A function that transforms its inputs into some smaller range of possible outputs.

Recipe for saving to a hash table:

- Take the key and plug it into the hash function, getting the hash code.
- Modulo that hash code by the length of the underlying array, getting an array index.
- Check if the array at that index is empty, if so, save the value (and the key) there.
- If the array is full at that index continue to the next possible position depending on your collision strategy.

Recipe for retrieving from a hash table:

- Take the key and plug it into the hash function, getting the hash code.
- Modulo that hash code by the length of the underlying array, getting an array index.
- Check if the array at that index has contents, if so, check the key saved there.
- If the key matches the one you're looking for, return the value.
- If the keys don't match, continue to the next position depending on your collision strategy.