Exploration: Introduction to Maps and Hash Tables

Introduction

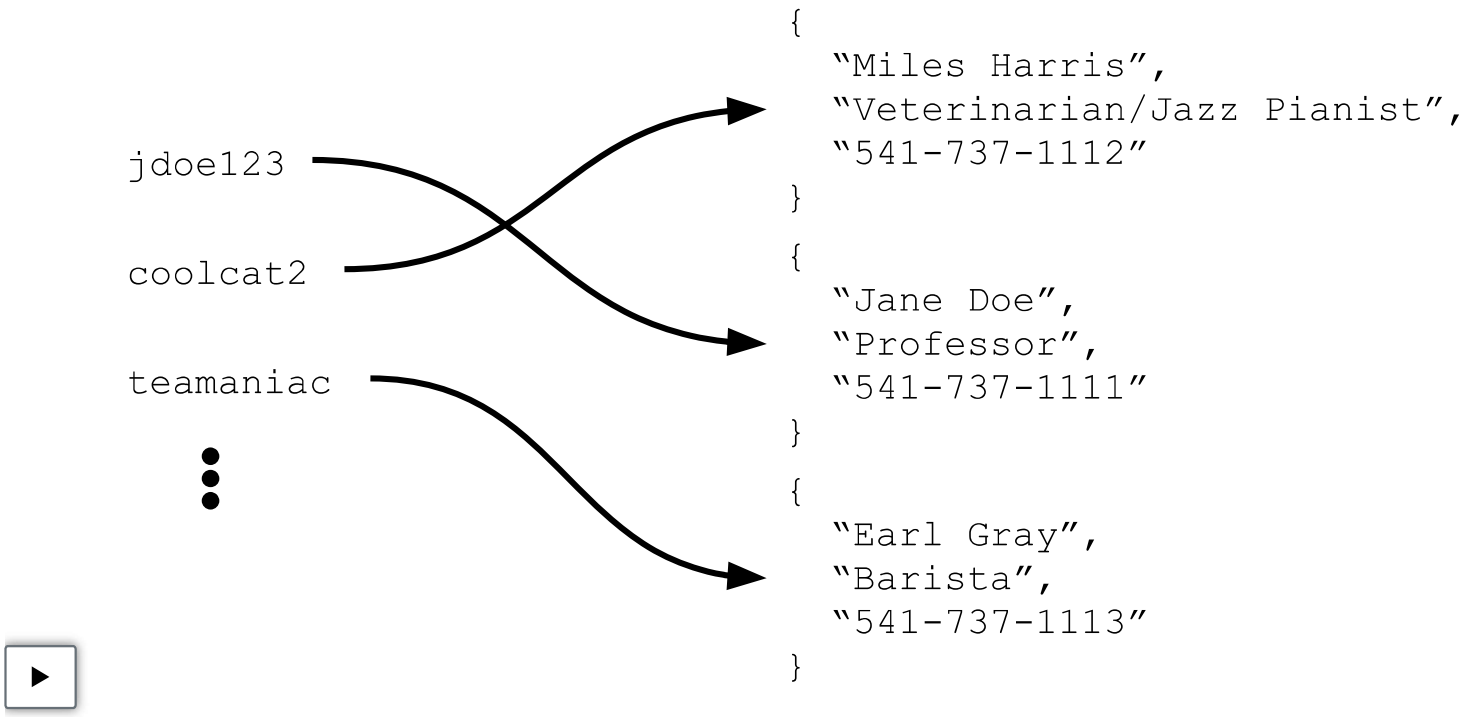
Trees offer overall good performance on a wide variety of operations. But sometimes, all we really want to do is add elements to a collection and look them up.

When insertion, lookup, and removal are the only operations we need, we can use the map data type.

A map is also known as a dictionary or an associative array. It is built into Python, and you have used it using the syntax `{key1: value1, key2: value3 …}`. This section will explore how this is implemented, and some ways we can optimize them.

Maps

Let's say we are building a web application. Our user data may be represented in a map, where the key was the username or email address, and the value was all the other data about each user (phone number, address, past purchases, contacts, etc.):



A map tying keys to more complex data values

Or, let's say we are counting instances of all the words in a document. We can use a map where the key is the unique word, and the value is the number of times that word occurs. We've already seen data structures that would allow us to implement a map structure:

- We could use a simple array, storing key/value structs.
  - This would give us O(n) insertions and lookups (or O(log n) lookups, if we ordered the array by key).
- We could use an AVL tree, also storing key/value structs.
  - This would give us O(log n) insertions and lookups.

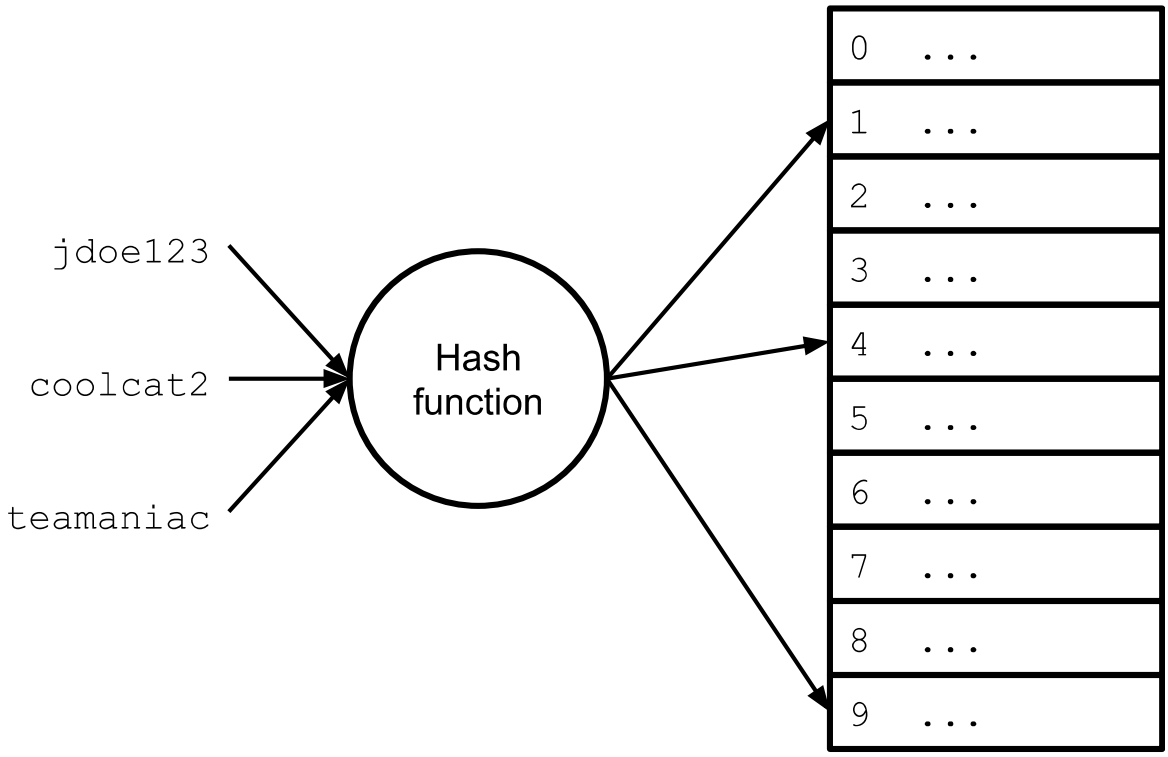
Using a hash table, we can better than this.

Hash Tables

A hash table is like an array, with a few important differences:

- Elements can be indexed by values other than integers.
- More than one element may share an index.

The key to implementing a hash table is a hash function, which is a function that takes values of some type (e.g. string, struct, double, etc.) and maps them to an integer index value. We can then use this value to both store and retrieve data from an actual array, e.g.:



Inputs passing through a hash function being mapped to array indexes

▶

Often, the hash function computes an index in two steps:

```
hash = hash_function(key)
index = hash % array_size
```

The first step computes a value based on the key. However, we may not always be working with arrays of a size that exactly matches the values the hash function can produce. This means we need to use the `%` operator to make sure the value gets assigned to an index that actually exists in our array.

When choosing or designing a hash function, there are a few properties that are desirable:

- **Determinism** – a given input should always map to the same hash value.
- **Uniformity** – the inputs should be mapped as evenly as possible over the output range. A non-uniform function can result in many collisions, where multiple elements are hashed to the same array index. We'll look more at this later.
- **Speed** – the function should have a low computational burden.

For example, if we were hashing strings, a simple hash function might sum the ASCII values of the characters, e.g.:

```
"eat" ⇨ 'e' + 'a' + 't' = 101 + 97 + 116 = 314
```

An operation like this is known as a folding operation. This can have problems, though:

```
"eat" ⇨ 'e' + 'a' + 't' = 101 + 97 + 116 = 314
"ate" ⇨ 'a' + 't' + 'e' = 97 + 116 + 101 = 314
"tea" ⇨ 't' + 'e' + 'a' = 116 + 101 + 97 = 314
```

Passing these three strings through this hash function results in the same integer value. For instances like this, we can use a shifting operation, which modifies the individual components of a folding operation based on their position (e.g. multiply by 2^0, 2^1, 2^2, 2^3, …).

```
"eat" ⇨ 'e' + 'a' + 't' = 1* 101 + 2* 97 + 4* 116 = 759
"ate" ⇨ 'a' + 't' + 'e' = 1* 97 + 2* 116 + 4* 101 = 733
"tea" ⇨ 't' + 'e' + 'a' = 1* 116 + 2* 101 + 4* 97 = 609
```

Of course, these hash functions are very simplified hash functions that we'd (hopefully) never use in real life.

Here's an example of a well-known and widely-used hash function, the DJB2 hash function (for strings):

```
def hash_djb2(s):
    hash = 5381
    for x in s:
        # hash * 33 + c (bitshift left 5 places = * 32)
        hash = ( (hash << 5) + hash ) + ord(x)
    return hash & 0xFFFFFFFF
```

▶

This function is simple and fast, though it could be faster (e.g. by processing multiple bytes at a time). It produces a good distribution. The numbers were chosen mainly by experimentation and are not terribly special.

Perfect and Minimally Perfect Hash Functions

A perfect hash function is one that results in no collisions; that is, every input gets a unique output.

A minimally perfect hash function is one that results in no collisions for a table size that equals exactly the number of elements.

For example, consider this collection of strings:

- “yummy”
- “delicious”
- “incredible”
- “fantastic”
- “exquisite”
- “nonpareil”

For this specific collection, the following function is minimally perfect:

```
def string_hash(my_string):
    return (ord(my_string[0]) - ord('a')) % 6
```

Specifically, we have all the values 0 through 5 covered:

```
string_hash("yummy") → 0       # 'y' - 'a' = 24
string_hash("delicious") → 3   # 'd' - 'a' = 3
string_hash("incredible") → 2  # 'i' - 'a' = 8
string_hash("fantastic") → 5   # 'f' - 'a' = 5
string_hash("exquisite") → 4   # 'e' - 'a' = 4
string_hash("nonpareil") → 1   # 'n' - 'a' = 13
```

Of course, in practice we usually don’t have such a nicely arranged situation, so it’s rare that our hash function will be minimally perfect. For example, let's say 2,450 keys are hashed into a million buckets. Even with a perfectly uniform random distribution, according to the **birthday problem (Links to an external site) (https://en.wikipedia.org/wiki/Birthday_problem)**, there is approximately a 95% chance of at least two of the keys being hashed to the same slot.

This means that we will most likely need to deal with collisions, which is where the hash function maps more than one element to the same index in our hash array.

We’ll look at two mechanisms for resolving hash conflicts, chaining and open addressing, in the next exploration.