Hashing

A data structure is a format for organizing data in an efficient way. Practically, we can split data structures into two things: the interface and the implementation.

The interface is like a contract that specifies how we can interact with the data structure - what operations we can perform on it, what inputs it expects, and what outputs we can expect.

For example, consider a dynamic array. The interface would include operations like appending, insertion, removal, updating, and more.

The implementation is the code that actually makes the data structure work. This is where the details of how the data is stored and how the operations are performed come into play. For example, the implementation of a dynamic array might involve allocating memory for the list, tracking the size, and rearranging the elements when an operation like remove is called.

The implementation can be quite complex. However, we don't need to worry about those details - we only need to understand the interface and how to use it properly.

All major data structures have built-in implementations in all major programming languages.

A hash function is a function that takes an input and deterministically converts it to an integer that is less than a fixed size set by the programmer.

What is the point of a hash function?

We know that arrays have $O(1)$ random access where the indices have to be integers. Since hash functions can convert any input into an integer, we can effectively remove the constraint of indices needing to be integers. When a hash function is combined with an array, it creates a hash table.

Typically keys have to be immutable but values can be anything.

A hash map is extremely powerful and frequently allows you to reduce the time complexity of an algorithm by a factor of $O(n)$ for a huge amount of problems.

A hash map is an unordered data structure that stores key-value pairs. A hash map can add and remove elements in $O(1)$, as well as update values associated with a key and check if a key exists also in $O(1)$. You can iterate over both the keys and values of a hash map, but the iteration won't necessarily follow any order.

Comparisons with arrays

The following operations are all $O(1)$ for a hash map:

- Add an element and associate it with a value
- Delete an element if it exists
- Check if an element exists

A hash map also has many of the same useful properties as an array with the same time complexity:

- Find the length/number of elements
- Updating values
- Iterate over elements

Even if your keys are integers and you could get away with using an array, if you don't know what the max size of your key is, then you don't know how large you should size your array. With hash maps, you don't need to worry about that, since the key will be converted to a new integer within the size limit anyways.

The biggest disadvantage of hash maps is that for smaller input sizes, they can be slower due to overhead. It is $O(1)$ but the constant can be like $O(10)$ in actuality because every key needs to go through a hash function and there can also be collisions.

Hash tables can also take up more space. Dynamic arrays are actually fixed-size arrays that resize themselves when they go beyond their capacity. Hash tables are also implemented using a fixed size array - the size is a limit set by the programmer. The problem is, resizing a hash table is much more expensive because every existing key needs to be rehashed, and also a hash table may use an array that is significantly larger than the number of elements stored, resulting in a huge waste of space.

Collisions

Prime numbers near significant magnitudes that are common to use for the size of hash tables are:

10,007
1,000,003
1,000,000,007

Sets

A set is similar to a hash table except that a set does not map its keys to anything. They are more convenient to use when you only care about checking if an element exists. You can add, remove, and check if an element exists in a set all in $O(1)$).

Sets don't track frequency.

Arrays as keys?

Usually keys must be immutable. Depending on the language, there are several ways to convery an array into a unique immutable key. In Python, tuples are immutable, so it's as easy as doing `tuple(arr)`. Another trick is to convert the array into a string, delimited by some character that is guaranteed to not show up in any element. For example, use a comma to separate integers. `[1, 51, 163] --> "1,51,163"`. Some languages like `C++` support mutable keys in their built-in implementations.

Interface guide

Python3

```
# Declaration: a hash map is declared like any other variable. The syntax is
{}
hash_map = {}

# If you want to initialize it with some key value pairs, use the following
syntax:
hash_map = {1: 2, 5: 3, 7: 2}
```

```
# Checking if a key exists: simply use the `in` keyword
1 in hash_map # True
9 in hash_map # False

# Accessing a value given a key: use square brackets, similar to an array.
hash_map[5] # 3

# Adding or updating a key: use square brackets, similar to an array.
# If the key already exists, the value will be updated
hash_map[5] = 6

# If the key doesn't exist yet, the key value pair will be inserted
hash_map[9] = 15

# Deleting a key: use the del keyword. Key must exist or you will get an
error.
del hash_map[9]

# Get size
len(hash_map) # 3

# Get keys: use .keys(). You can iterate over this using a for loop.
keys = hash_map.keys()
for key in keys:
    print(key)

# Get values: use .values(). You can iterate over this using a for loop.
values = hash_map.values()
for val in values:
    print(val)
```

Javascript

```
// Declaration: use the Map object
let hashMap = new Map();

// If you want to initialize it with some key value pairs, use the following
syntax:
let hashMap = new Map([
    [1, 2],
```

```
    [5, 3],
    [7, 2]
]);

// Checking if a key exists: use .has()
hashMap.has(1); // true
hashMap.has(9); // false

// Accessing a value given a key: use .get()
hashMap.get(5); // 3

// Adding or updating a key: use .set()
// If the key already exists, the value will be updated
hashMap.set(5, 6);

// If the key doesn't exist yet, the key value pair will be inserted
hashMap.set(9, 15);

// Deleting a key: use .delete()
hashMap.delete(9);

// Get size
hashMap.size; // 3

// Iterate over the key value pairs: use the following code
for (const [key, value] of hashMap) {
    console.log(key + " " + value);
}
```

Java

```
// Declaration: Java supports multiple implementations, but we will using
// the Map interface with the HashMap implementation. Specify the data type
// of the keys and values
Map<Integer, Integer> hashMap = new HashMap<>();

// If you want to initialize it with some key value pairs, use the following
syntax:
Map<Integer, Integer> hashMap = new HashMap<>() {{
    put(1, 2);
```

```
    put(5, 3);
    put(7, 2);
}};

// Checking if a key exists: use .containsKey()
hashMap.containsKey(1); // true
hashMap.containsKey(9); // false

// Accessing a value given a key: use .get()
hashMap.get(5); // 3

// Adding or updating a key: use .put()
// If the key already exists, the value will be updated
hashMap.put(5, 6);

// If the key doesn't exist yet, the key value pair will be inserted
hashMap.put(9, 15);

// Deleting a key: use .remove()
hashMap.remove(9);

// Get size
hashMap.size(); // 3

// Iterate over keys: use .keySet()
for (int key: hashMap.keySet()) {
    System.out.println(key);
}

// Iterate over values: use .values()
for (int val: hashMap.values()) {
    System.out.println(val);
}
```

C++

```
// Declaration: C++ supports multiple implementations, but we will be using
// std::unordered_map. Specify the data type of the keys and values.
unordered_map<int, int> hashMap;
```

```
// If you want to initialize it with some key value pairs, use the following
syntax:
unordered_map<int, int> hashMap = {{1, 2}, {5, 3}, {7, 2}};

// Checking if a key exists: use the following syntax:
hashMap.find(1) != hashMap.end(); // true
hashMap.find(9) != hashMap.end(); // false

// Accessing a value given a key: use square brackets, similar to an array.
hashMap[5]; // 3

// Note: if you were to access a key that does not exist, it creates the key
with a default value of 0.
hashMap[342]; // 0

// Adding or updating a key: use square brackets, similar to an array.
// If the key already exists, the value will be updated
hashMap[5] = 6;

// If the key doesn't exist yet, the key value pair will be inserted
hashMap[9] = 15;

// Deleting a key: use the .erase() method.
hashMap.erase(9);

// Get size
hashMap.size(); // 3

// Iterate over the key value pairs: use the following code.
// .first gets the key and .second gets the value.
for (auto const& pair: hashMap) {
    cout << pair.first << " " << pair.second << endl;
}
```