Maps / HashTables

Download Demo Code

Goals

- Define what a hash table is
- Describe how hash tables are implemented
- Describe what a hash function is, and what properties it should have
- Handle hashing collisions
- Identify time complexities of common hash table operations

Maps

Abstract Data Type for mapping key → value

```
let petsToAges = {
  "Whiskey": 6,
  "Fluffy": 2,
  "Dr. Slitherscale": 2
}
```

- Javascript: **Map** or **{}**
- Python: **dict**
- Ruby: **Hash**
- Java: **HashMap**
- Go: **map**

Typical API

set(key, val)
Sets **key** to **val**

get(key)
Retrieve values for **key**

delete(key)
Delete entry for **key**

has(key)
Is there an entry for **key**?

keys()
Iterable of keys

values()
Iterable of values

entries()
Iterable of key/value pairs

Simple Implementation

```
class SimpleMap {
  constructor() { this._items = []; }

  set(k, v) { this._items.push([k, v]); }

  get(k) {
    let kv = this._items.find(kv => k === kv[0]);
    return kv ? kv[1] : undefined;
  }

  has(k) {
    return this._items.find(kv => k === kv[0]) !== undefined;
  }

  delete(k) {
    let i = this._items.findIndex(kv => k === kv[0]);
    if (i !== -1) this._items.splice(i, 1);
  }

  keys()    { return this._items.map(kv => kv[0]); }
  values()  { return this._items.map(kv => kv[1]); }
  entries() { return this._items; }
}
```

Runtime for our simple implementation:

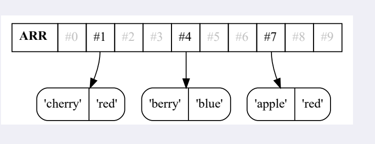
Operation	Runtime
set	O(1)
get	O(n)
has	O(n)
delete	O(n)
keys	O(n)
values	O(n)
entries	O(n)

We can do better with a different implementation!

Hash Tables

```
let fruits = {"apple": "red",
              "berry": "blue",
              "cherry": "red"}
```

- It'd be awesome to keep this in some sort of magic array
  - Get *O(1)* time for many operations



But how could we know that "apple" is index #7?

Hashing

We can hash a string to a number using *charCode*

```
str: a   p   p   l   e
cC : 97  112 112 108 101 = 530
```

We could store "apple" in index #530!

```
function hash(key) {
  return Array.from(key).reduce(
    (accum, char) => accum + char.charCodeAt(),
    0
  );
}
```

- We might get huge index #s, though
- For "supercalifragilisticexpialadocious", we'd get #3,747
- If we only needed to map 10 different words, we'd waste space
- Solution: Use modulo (`%`) to truncate: `hash % array.length`

```
function hash(key, arrayLen) {
  hash = Array.from(key).reduce(
    (accum, char) => accum + char.charCodeAt(),
    0
  );

  return hash % arrayLen;
}
```

- This would hash "act" and "cat" to the same number
- We'll use "Horner's Method" to make order meaningful:
  - For each letter, we add `n * currHash + currLetter`

```
function hash(key, array_len) {
  // Prime number to use with Horner's method
  const H_PRIME = 31;

  let numKey = Array.from(key).reduce(
    (accum, char) => accum * H_PRIME + char.charCodeAt(),
    0
  );

  return numKey % array_len;
};
```

Note: Why 31?

Prime numbers tend to be used to make hashes — and particular prime numbers are better than others. The explanation is interesting, but delves deeply into math theory, and is not something most developers will ever learn. If you're interested, though: Why Do Hash Functions Use Prime Numbers?

Runtime of Hash

- Amount of work to hash key isn't related to # items in map
- In our implementation: it is related to length of input string
  - So we can call it ***O(k)***, where ***k*** is #-chars-in-string
- Real-world versions often use part of string *(eg first 100 chars)*
  - These then could be ***O(1)***, as length-of-key doesn't affect worst case
- We'll assume hash is ***O(1)*** in discussion of runtime for hash tables

Fast Hashes vs Crypto Hashes

Hash functions for hash tables, prioritize:

- speed (must be fast!)
- wide distribution (spread out values so there are fewer collisions)

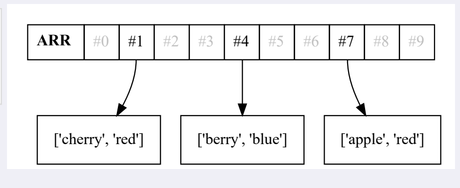
For cryptologic hashes, like SHA or Bcrypt, prioritize:

- dfficulty of reversing output

For crypto uses, always use a proven crypto hash, not your own!

Hash Table

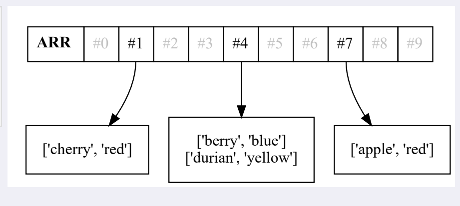
```
apple → 7
berry → 4
cherry → 1
```



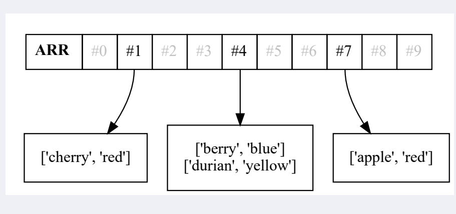
Oh no! Two keys hash same?

```
apple → 7
berry → 4
cherry → 1
durian → 4
```

Solution: Each *bin* is array of `[key, val]` s

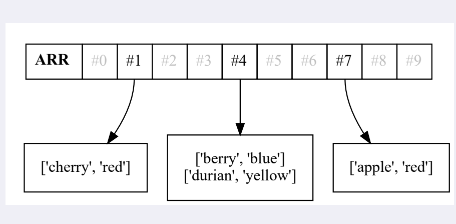


HashTable *set(key, val)*



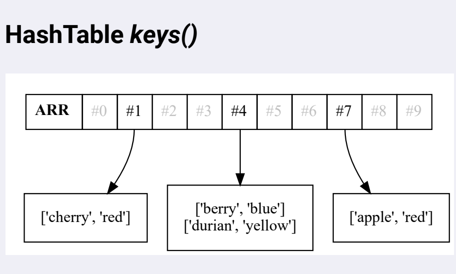
- Hash key
- If bin is empty: set to `[key, val]`
- Else: add `[key, val]` to end

HashTable *get(key)*



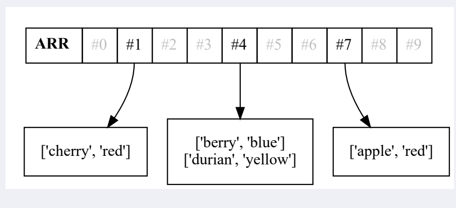
- Hash key
- If bin is empty: return ***undefined***
- Search array, returning value if found
- If not in array, return ***undefined***
- ***has()***: same idea, returns true/false

HashTable *keys()*



- Loop over bins
- For each bin, loop over pairs
- ***values()*** and ***entries()*** are same idea

HashTable *delete(key)*



- Hash key
- If bin is empty: return
- Search array for index of item
- Splice array to remove item

HashTable Runtimes

set	O(1)
get, has	mostly O(1)
delete	mostly O(1)
keys, values, entries	O(n)

WTF Does "Mostly" Mean???

- Fundamentally, hash tables can be ***O(1)***
  - If we don't have collision & array is right size
- You can get close to ***O(1)*** by:
  - Choosing array size large enough to minimize collisions
  - Choosing hash function that spreads keys evenly in array
- If you have predictable number of collisions, it can be ***O(1)***
  - Remember: ***O(3)*** is the same as ***O(1)*** in runtime!

Resizing

- To ensure efficiency, good implementation shrink/grow array
  - Often aiming to keep it ~75% occupied
- This means *some* ***.set()*** and ***.delete()*** calls will take longer
  - If shrink/grown by proportion *(eg, double/halve)*, will be "amortized ***O(1)***"

Collisions

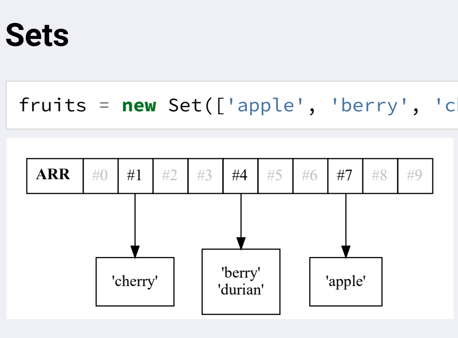
- Our first implementation made each bin (spot in array) an array
- This is a common implementation; it's called "chaining"
- There's another possibility

Open Addressing

- We can make each bin just a single `[key, value]` pair
- If collision: look at the "next" place
  - This can be the next bin (this is "linear probing")
  - Or there are smarter algorithms to reduce clumping
- We should keep array size large enough to minimize when this happens
- If we do and we have a good hash function, we can get amortized ***O(1)***

Sets

```
fruits = new Set(['apple', 'berry', 'cherry', 'durian'])
```



- A ***Set*** is just a ***Map*** without values
- Same runtime characteristics

Javascript Types

Map

- Built-in type for mapping
- Keys can be any type
  - Retrieval uses **===** to match
- Keeps keys in order of insertion
- Amortized ***O(1)*** for set/get/delete

```
let fruits = new Map(
  [["apple", "red"],["berry", "blue"]])

fruits.set("cherry", "red")

// some methods return map, so can chain
fruits.set("cherry", "red")
      .set("durian", "yellow")
      .delete("apple")

let berry_color = fruits.get("berry")
```

Object

- Generic object; can use for mapping
- Prior to ***Map*** (2015), was only way!
- Keys can only be strings or numbers
  - Numbers stringified: ***1*** → ***"1"***
- Keeps keys in order of insertion
- Amortized ***O(1)*** for set/get/delete
- Better to use ***Map*** for mapping

```
let fruits = {"apple": "red",
              "berry": "blue"}

fruits.cherry = "red"
fruits["durian"] = "yellow"

let berry_color = fruits.berry
let cherry_color = fruits["cherry"]
```

Keys can be a few other less common things, such as Javascript "Symbol" types, though these are uncommon for use in mapping (this is more common when making special methods for OO). The ordering of keys can also at times be a bit complex when you have different types of keys.

Set

- Built-in type for sets
- Keys can be any type
  - Retrieval uses **===** to match
- Keeps keys in order of insertion
- Amortized ***O(1)*** for set/get/delete

```
let fruits = new Set(["apple", "berry"])

fruits.add("cherry")
fruits.has("apple")  // true
```

Python Types

Dictionary

- Built-in type for mapping
- Keys can be any *immutable* type
- Keeps keys in order of insertion *(Python > 3.6)*
- Amortized ***O(1)*** for set/get/delete

```
fruits = {"apple": "red", "berry": "blue"}

also_can = dict(apple="red", berry="blue")

fruits["cherry"] = "red"

fruits["berry"]       # error if not there
fruits.get("cherry")  # or None

# dict comprehension
{x: x * 2 for x in numbers if x > 5}
```

Set

- Built-in type for sets
- Keys can be any *immutable* type
- Key order not guaranteed
- Amortized ***O(1)*** for set/get/delete
- Has awesome built-in set operations
  - Union, intersection, symmetric difference, subtraction
  - For JS, can get these with awesome ***lodash*** library

```
moods = {"happy", "sad", "grumpy"}

dwarfs = set(["happy", "doc", "grumpy"])

# union, intersection, and symmetric diff:
moods | dwarfs  # {happy, sad, grumpy, doc}
moods & dwarfs  # {happy, grumpy}
moods ^ dwarfs  # {sad, doc}

# subtraction
moods - dwarfs  # {sad}
dwarfs - moods  # {doc}

# set comprehension
{n for n in some_list if n > 10}
```

Frozenset

- Same as ***set()***, but immutable
  - Useful to use as a key in a ***dict***
- Same runtime, same API, same set functions

```
moods = frozenset(["happy", "sad", "grumpy"])

dwarfs = frozenset(["happy", "doc", "grumpy"])

# union, intersection, and symmetric diff:
moods | dwarfs  # {happy, sad, grumpy, doc}
moods & dwarfs  # {happy, grumpy}
moods ^ dwarfs  # {sad, doc}

# subtraction
moods - dwarfs  # {sad}
dwarfs - moods  # {doc}
```

Learning More

- Awesome writeups from Base CS:
  - Taking Hash Tables Off the Shelf
  - Hashing Out Hash Functions
- Python's method for ordered dictionaries
- Perfect hash tables
  - If you know your keys in advance, you can have a hash table without chains *or* open addressing (just simple bins)
  - There are algorithms that can discover a "perfect hash function" for your keys that produce a unique hash for each key
  - Useful for small, fast, simple lookup tables than don't change