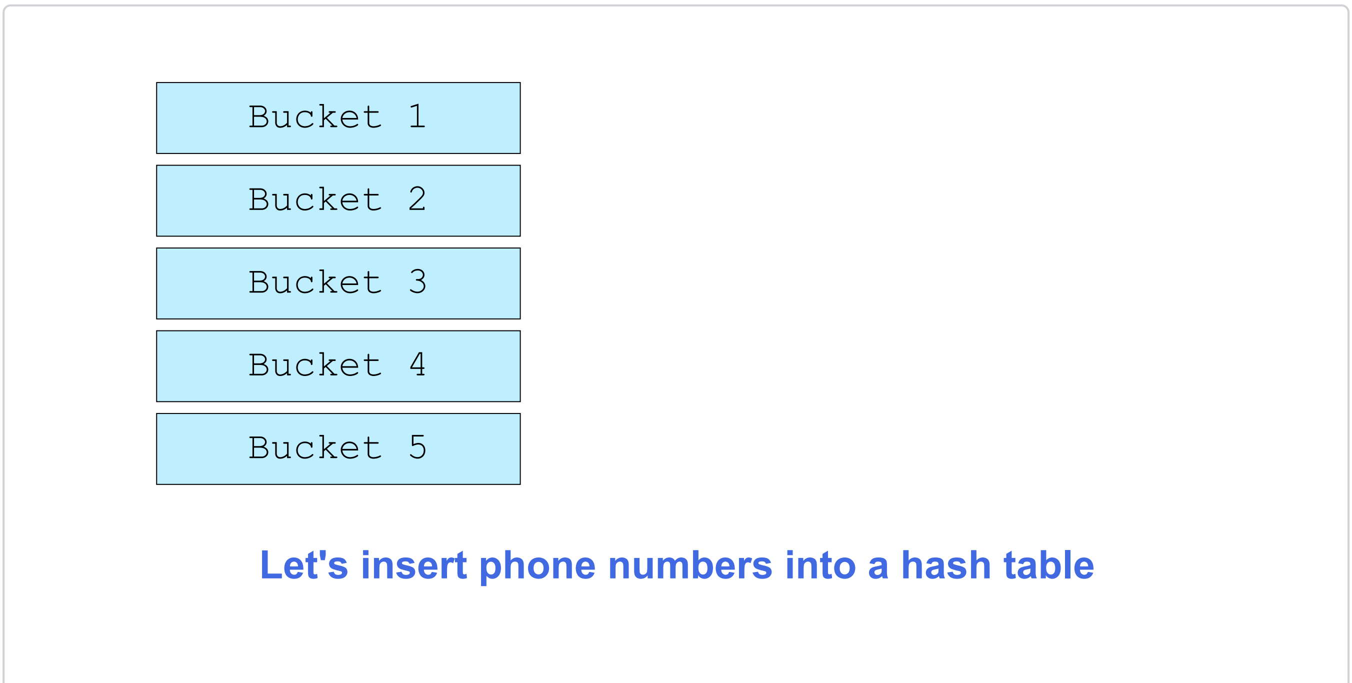
Hash Table

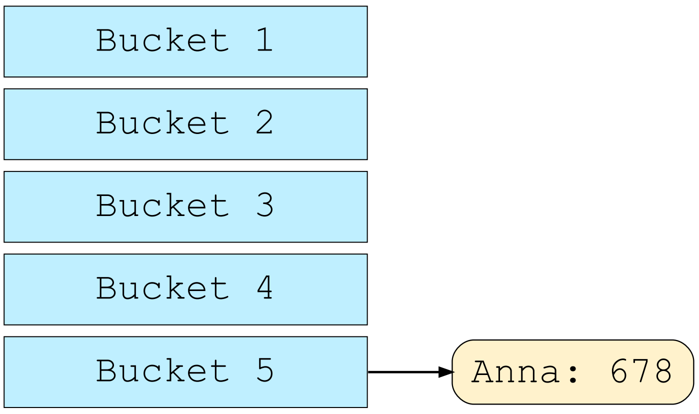
What is a Hash Table?

A Hash table is a data structure that is used to implement an associative array of a key and value. But we said the same thing about the Dictionary. How is different from a dictionary?

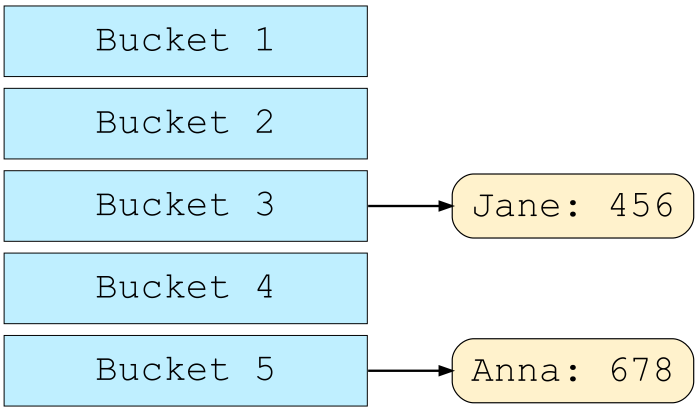
A dictionary is a 1:1 map of a key/value pair. Hashtable is a more efficient way of storing key/value pairs. It is designed around an array where each array cell is called a ***bucket***. Each bucket contains a list of key/value pairs. When a key/value pair is inserted into a hashtable, we first compute it's destination bucket. We determine the bucket by applying a special function on the key. This special function is known as the ***hash function***. Every hash function returns a number based on the key. This number is the bucket Id. We then go to the bucket and append this key/value pair to the list of key/value pairs in the bucket.

Here's a quick visualization to look at the hash table (Our Hash Table has 5 buckets).

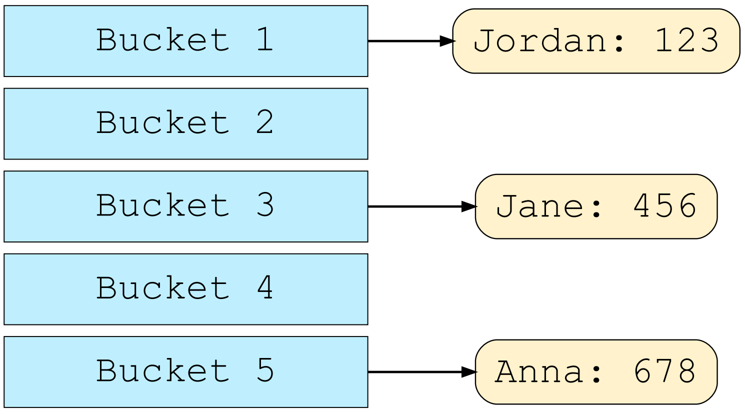




Insert Anna's Phone: 678.
Key 'Anna' maps to bucket 5.



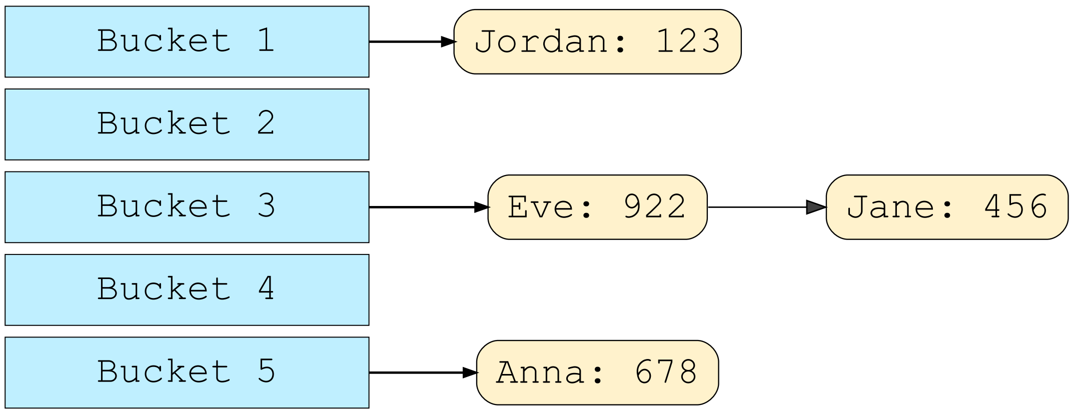
Insert Jane's Phone: 456.
Key 'Jane' maps to bucket 3.



Insert Jordan's Phone: 123.

Key 'Jordan' maps to bucket 1.

4 of 5



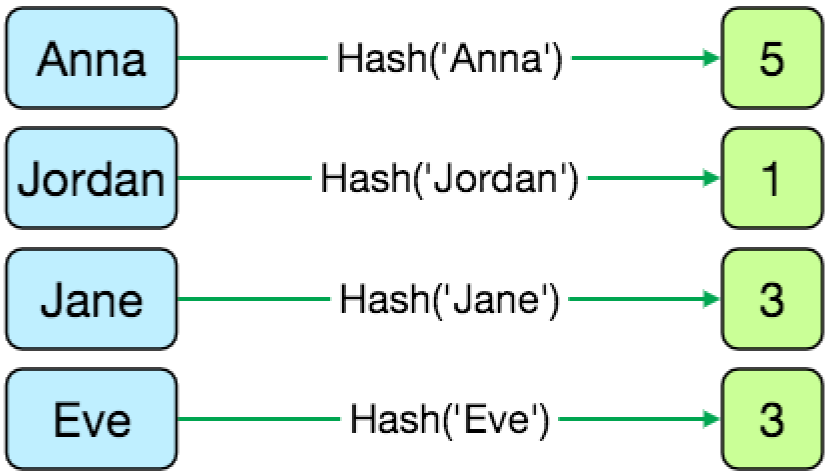
Insert Eve's Phone: 922.
Key 'Eve' maps to bucket 3.

5 of 5

When we have to add a key/value pair to a bucket that already has one or more key value pairs, we say that a ***collision*** has occurred. Whenever a collision occurs, we apply a collision resolution strategy. We'll look into these strategies in a bit. However, lets first focus on the the Hash functions.

Hashing and Hash Function

As we've already seen, Hash tables rely on a hash function to convert keys into a bucket index. The implementations of Hash functions range from simple functions to very sophisticated. The goal of a production quality hash function is to avoid collisions as much as possible. Such hash functions are called uniform hash functions. More formally, a ***uniform hash*** function is a function that maps the expected inputs into available buckets as evenly as possible. If you look at the figure below, it's pretty clear that our hash function is not a uniform one.



Hash function used in our Hash Map Visualization

To keep things simple, we are going to use a very basic hash function but it should give you an idea about how hash functions are implemented. Here's the definition of our Hash Table.

```
function HashTable() {
  this._bucketSize = 23;
  this._buckets = [];
  this._buckets.length = this._bucketSize;
}
```

If you notice, we have set the bucket count to 23. The total number of buckets is usually a prime number (and usually a relatively large one than our pick 23). Now we know that we will have a total of 23 buckets. Let's implement our very basic hash function.

```
HashTable.prototype.computeHash = function(key) {
  var total = 0;
  for (var i = 0; i < data.length; ++i) {
    total += data.charCodeAt(i);
  }

  return total % this._bucketSize;
}
```

We calculate the sum of the character codes of the data and then make sure that the final hash is less than the number of buckets. Based on this hash function, let's implement our Insert function (It's incomplete without handling collisions but we'll get there).

```
HashTable.prototype.put = function(key, value) {
  var hash = this.computeHash(key);

  if (this._buckets[hash] != undefined) {
    throw 'We are not handling collisions yet";
  }

  this._buckets[hash] = value;
}
```

If you look at the above function, it's clear that we can only store one key/value pair in a bucket. In addition, if you notice closely, there's also a bug in the above code. We are not saving the key. We'll run into problems because a hash function can return the same hash for many keys. Hence, in our example above, if you add the phone number of Jane, and the try to get the phone number of Eve, we will return you Jane's number. Hence, the first learning is that we need to store all keys along with their values. We will fix that when we implement the code again but first let's look at collision resolution techniques.

Collision Resolution in Hash Table

There are two basic techniques to handle collisions in a hash table.

1. Separate chaining
2. Open addressing

Let's look at both of them in a little bit more detail.

Separate Chaining

In this method, each bucket contains a key/value pair list itself. In fact, the first visualization we saw (with a collision with Eve and Jane), we handled the collision by separate chaining. Separate chaining is easy to implement and understand. If the hash function is almost uniform and the bucket size is large enough, collisions would be rare and a few buckets with chains of 3 or 4 keys/value pairs is not a big issue. However, if the collisions are frequent, then some buckets would be heavily loaded and lookups within a bucket can be slow (which would significantly reduce the efficacy of our hash table).

Open Addressing

Open Addressing

In this method, instead of having a list within a list, we have a single array of buckets and all key/value pairs are stored in a bucket. The underlying assumption is that the bucket count is large enough to accommodate all key/value pairs. Insert works by looking at the bucket returned by the hash function. If the bucket is occupied, it starts probing for the next bucket using a ***Probing Sequence***.

One most common probing sequence is called ***Linear Probing*** where we start looking for next free bucket by adding a fixed integer to the current bucket. e.g. If hash function returned bucket5 but bucket5 is occupied, then we will start looking at bucket6, bucket7 … until we find a free bucket. You can see that with this approach we cannot handle key/value pairs larger than the number of buckets. If we have to probe many times before we find a slot, it will not only slow down the insertions, it will also slow down the lookups.

Which collision resolution technique is better?

Both collision resolution techniques that we discussed have pros and cons. You need to pick the technique based on the amount and nature of the data. Typically, hash tables implement a re-hashing when collisions go beyond a certain threshold. However, it's a costly operation. Most hash table implementations would let you specify the number of buckets at its creation. This helps in avoiding collisions and paying the cost of re-hashing.

We'll implement both techniques for our learning here.

Implementing Hash Table with Separate Chaining

Javascript has a built-in object so instead of storing a linked list or array in each bucket, we are going to store an object and use that object to store key/value pairs.

Here's how we will implement our put function using separate chaining

```
HashTable.prototype.put = function(key, value) {
  var keyType = typeof(key);
  if (keyType !== 'string' && keyType !== 'number') {
    throw 'Only string or number keys are supported';
  }

  var hash = this.computeHash(key);


  if (this._buckets[hash] === undefined) {
    this._buckets[hash] = {};
  }

  var chain = this._buckets[hash];

  if (chain.hasOwnProperty(key)) {
    throw 'Duplicate key is not allowed';
  }

  chain[key] = value;
}
```

and here's the corresponding get function

```
HashTable.prototype.get = function(key) {
  var keyType = typeof(key);
  if (keyType !== 'string' && keyType !== 'number') {
    return undefined;
  }

  var hash = this.computeHash(key);

  if (this._buckets[hash] === undefined) {
    return undefined;
  }

  var chain = this._buckets[hash];

  if (chain.hasOwnProperty(key)) {
    return chain[key];
  }

  return undefined;
}
```

>Run the following program to see Hash Table implemented with separate chaining.

JavaScript

HTML

CSS (SCSS)

```
var contacts = new HashTable();
contacts.put("Anna", 678);
contacts.put("Jordan", 123);
console.log('Anna\'s Phone: ' + contacts.get('Anna'));
console.log('Jordan\'s Phone: ' + contacts.get('Jordan'));
console.log('Frank\'s Phone: ' + contacts.get('Frank'));
```

Console

Clear

```
Anna's Phone: 678
Jordan's Phone: 123
Frank's Phone: undefined
```

Implement Hash Table with Open Addressing

We'll use linear probing to implement collision resolution. Here's the code for implementing put using linear probing.

```
HashTable.prototype.put = function(key, value) {
  var keyType = typeof(key);
  if (keyType !== 'string' && keyType !== 'number') {
    throw 'Only string or number keys are supported';
  }

  var hash = this.computeHash(key);

  if (this._buckets[hash] === undefined) {
    // Yay No collision found
    this._buckets[hash] = {};
    this._buckets[hash][key] = value;
    return;
  } else if (this._buckets[hash].hasOwnProperty(key)) {
    // Duplicate Key
    throw 'Duplicate Key is not allowed';
  }

  // Collision found.
  // Let's probe for the next available slot
  var bucketId = hash + 1;

  do {
    if (bucketId >= this._bucketSize) {
      // Reached the end.
      // Start from the beginning
      bucketId = 0;
    }

    if (this._buckets[bucketId] === undefined) {
      // Found an empty slot
      this._buckets[bucketId] = {};
      this._buckets[bucketId][key] = value;
      return;
      return;
    }

    bucketId++;
  } while (bucketId != hash);

  // Couldn't find any free slots.
  throw 'Hash Table is full. Rehashing needed';
}
```

and here's the corresponding get function.

```
HashTable.prototype.get = function(key) {
  var keyType = typeof(key);
  if (keyType !== 'string' && keyType !== 'number') {
    return undefined;
  }

  var hash = this.computeHash(key);

  if (this._buckets[hash] === undefined) {
    return undefined;
  } else if (this._buckets[hash].hasOwnProperty(key)) {
    // Key found. Return value
    return this._buckets[hash][key];
  }

  // Possible Collision.
  // Iterate through the table using the
  // probing sequence used by the put function
  var bucketId = hash + 1;

  do {
    if (bucketId >= this._bucketSize) {
      // Reached the end.
      // Start from the beginning
      bucketId = 0;
    }

    if (this._buckets[bucketId] === undefined) {
      // Found an empty slot
      return undefined;
    } else if (this._bucekts[bucketId].hasOwnProperty(key)) {
      // Key found. Return value
      return this._buckets[hash][key];
    }

    bucketId++;
  } while (bucketId != hash);

  // Couldn't find the key and exhausted the
  // whole hash table.
  return undefined;
}
```

>Run the following program to see Hash Table implemented with open addressing (linear probing).

JavaScript

HTML

CSS (SCSS)

```
var contacts = new HashTable();
contacts.put("Anna", 678);
contacts.put("Jordan", 123);
console.log('Anna\'s Phone: ' + contacts.get('Anna'));
console.log('Jordan\'s Phone: ' + contacts.get('Jordan'));
console.log('Frank\'s Phone: ' + contacts.get('Frank'));
```

Console

Clear

```
Anna's Phone: 678
Jordan's Phone: 123
Frank's Phone: undefined
```

Summary

Hash Table is a data structure used to implement key value pairs.

Hash Table consists of a buckets (slots) and a hashing function maps the keys to slots.

Hash function can have collisions. When there is a collision, there are strategies to resolve to collision.

Two most common stragies for conflicit resolution are called separate chaining and open addressing.