**LEARN HASH MAPS: JAVASCRIPT**

**Intro to Hash Maps**

*Hash maps* are data structures that serve as efficient key-value stores. They are capable of assigning and retrieving data in the fastest way possible. This is because the underlying data structure that hash maps use is an array.

A value is stored at an array index determined by plugging the key into a hash function. Because we always know exactly where to find values in a hash map, we have constant access to any of the values it contains.

This quick access to values makes a hash map a good choice of data structure whenever we need to store a lot of values but need fast look-up time.

In JavaScript, objects are often used to map keys to values as in a hash map, but in this lesson, you’ll create your own implementation of a hash map by building out a HashMap class. You’ll build methods to hash and compress a given key, assign an index at which to store a value, and retrieve that value.

To implement a hash map, the HashMap constructor method will create an empty array that will hold values. A hashing function will return an index in the array where the value will be stored. While going through the following exercises, remember that the purpose of this lesson is to gain a deeper understanding of the data structure rather than creating a production-quality data structure.

**Instructions**

**1.**

The hash map class we will be completing is stored in the **HashMap.js** file. Look over the code you’ve been given, the constructor method.

Hash maps are based on arrays, so whenever we create a new hash map instance, we’ll create an array filled with null values that can be replaced with key-value pairs later.

**HashMap.js**

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

}

module.exports = HashMap;

**Hashing**

The *hashing function* is the secret to efficiently storing and retrieving values in a hash map. A hashing function takes a key as input and returns an index within the hash map’s underlying array.

This function is said to be *deterministic*. That means the hashing function must always return the same index when given the same key. This is important because we will need to hash the key again later to retrieve the stored value. We won’t be able to do this unless our hashing function behaves in a predictable and consistent way.

Getting an integer representing an index can be done by summing up each character code of the key (as a numeric value) with the running total of the previously summed character codes.

The hashing function should follow this logic:

declare hashCode variable with value of 0

for each character in the key

add the sum of the current character code value and hashCode to hashCode

return hashCode

Adding the sum of hashCode and the character code to the hashCode again creates a deterministic and also non-reversible implementation of a hashing function. This avoids generating a duplicate index if keys have the same characters in different orders, such as bat and tab.

**Instructions**

**1.**

Create a HashMap method, .hash(), with key as a parameter. This method will take a string and use it to generate an index in the hash map’s internal array.

Checkpoint 2 Passed

**2.**

To generate an index for each key-value pair, we’ll calculate a number based on the characters in the input string. Declare a variable whose value can be changed within .hash() called hashCode. Assign it an initial value of 0.

This variable will keep a running total of character codes.

Checkpoint 3 Passed

Hint

The let keyword creates a variable whose value can be reassigned later.

**3.**

After declaring hashCode create a for loop that loops over each character in key.

Checkpoint 4 Passed

Hint

This is an example of a for loop in JavaScript:

const recipes = ['jollof', 'adobo', 'frybread'];  
   
for(let i = 0; i < recipes.length; i++) {  
  console.log(`Serving up ${recipes[i]}!`);  
}

**4.**

Inside of the for loop convert each character in key to an integer using the JavaScript string method .charCodeAt().

This method only works on strings and converts a character at a specific index into an integer between 0 and 65535. This integer represents the equivalent [Unicode](https://developer.mozilla.org/en-US/docs/Glossary/Unicode) value of that character.

To use .charCodeAt() call it on a string with the index of the character you want the character code of:

// The code below will return the character code of 'H'  
'Hello world!'.charCodeAt(0) // => 72

Add the result of calling .charCodeAt() on the current character of key and hashCode to the hashCode variable.

Outside of the for loop, return the finished hashCode.

Checkpoint 5 Passed

Hint

Using the string method .charCodeAt(), your code should add and assign the character code of i to hashCode. Your solution should look like this:

hashCode += hashCode + key.charCodeAt(i);

The hashCode += allows the hashing function to avoid generating duplicate hashCodes if keys have the same characters in different orders, such as bat and tab.

**HashMap.js**

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode;

  }

}

module.exports = HashMap;

**Hashing**

The *hashing function* is the secret to efficiently storing and retrieving values in a hash map. A hashing function takes a key as input and returns an index within the hash map’s underlying array.

This function is said to be *deterministic*. That means the hashing function must always return the same index when given the same key. This is important because we will need to hash the key again later to retrieve the stored value. We won’t be able to do this unless our hashing function behaves in a predictable and consistent way.

Getting an integer representing an index can be done by summing up each character code of the key (as a numeric value) with the running total of the previously summed character codes.

The hashing function should follow this logic:

declare hashCode variable with value of 0

for each character in the key

add the sum of the current character code value and hashCode to hashCode

return hashCode

Adding the sum of hashCode and the character code to the hashCode again creates a deterministic and also non-reversible implementation of a hashing function. This avoids generating a duplicate index if keys have the same characters in different orders, such as bat and tab.

**Instructions**

**1.**

Create a HashMap method, .hash(), with key as a parameter. This method will take a string and use it to generate an index in the hash map’s internal array.

Checkpoint 2 Passed

**2.**

To generate an index for each key-value pair, we’ll calculate a number based on the characters in the input string. Declare a variable whose value can be changed within .hash() called hashCode. Assign it an initial value of 0.

This variable will keep a running total of character codes.

Checkpoint 3 Passed

Hint

The let keyword creates a variable whose value can be reassigned later.

**3.**

After declaring hashCode create a for loop that loops over each character in key.

Checkpoint 4 Passed

Hint

This is an example of a for loop in JavaScript:

const recipes = ['jollof', 'adobo', 'frybread'];  
   
for(let i = 0; i < recipes.length; i++) {  
  console.log(`Serving up ${recipes[i]}!`);  
}

**4.**

Inside of the for loop convert each character in key to an integer using the JavaScript string method .charCodeAt().

This method only works on strings and converts a character at a specific index into an integer between 0 and 65535. This integer represents the equivalent [Unicode](https://developer.mozilla.org/en-US/docs/Glossary/Unicode) value of that character.

To use .charCodeAt() call it on a string with the index of the character you want the character code of:

// The code below will return the character code of 'H'  
'Hello world!'.charCodeAt(0) // => 72

Add the result of calling .charCodeAt() on the current character of key and hashCode to the hashCode variable.

Outside of the for loop, return the finished hashCode.

Checkpoint 5 Passed

Hint

Using the string method .charCodeAt(), your code should add and assign the character code of i to hashCode. Your solution should look like this:

hashCode += hashCode + key.charCodeAt(i);

The hashCode += allows the hashing function to avoid generating duplicate hashCodes if keys have the same characters in different orders, such as bat and tab.

**HashMap.js**

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode;

  }

}

module.exports = HashMap;

**Compression**

The current hashing function will return a wide range of integers — some of which are not indices of the hash map array. To fix this, we need to use *compression*.

Compression means taking some input and returning an output only within a specific range.

In our hash map implementation, we’re going to have our hashing function handle compression in addition to hashing. This means we’ll add an additional line of code to compress the hashCode before we return it.

The updated .hash() should follow these steps:

initialize hashCode variable to 0

for each character in the key

add the character code and hashCode to hashCode

return compressed hashCode

**Instructions**

**1.**

Currently, our .hash() method is generating an integer representing an index but it’s not guaranteed that this index will be within the bounds of the hash map’s array.

To do this, we’ll use *modular arithmetic*. Because modular arithmetic prevents a value from growing larger than some limit, it’s a common solution when we want a value to “wrap around”.

After the for loop in the .hash() of HashMap, compress the value stored in hashCode by using modular arithmetic to return the remainder of dividing hashCode by the length of the hash map.

Checkpoint 2 Passed

Hint

The modulo operator, %, in JavaScript returns the remainder when dividing two numbers.

For example:

11 % 2 // => 1

**2.**

Check your work. Save a new HashMap instance with a size of 3, in a constant myHashMap and use the new .hash() to log the result of hashing the key 'id'. Hash and log 'id' again. Are the logged values the same or are they different?

**HashMap.js**

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

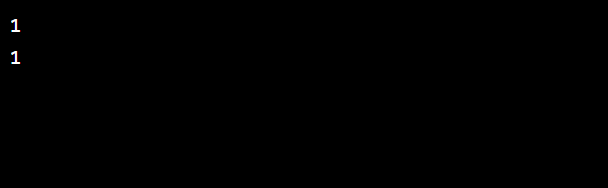
}

module.exports = HashMap;

const myHashMap = new HashMap(3);

console.log(myHashMap.hash('id'));

console.log(myHashMap.hash('id'));

****

**Assign**

We now have everything we need to find a place in the hash map array to store a value. The only thing left to do is assign the value to the index we generated. A method, .assign() will handle the logic needed to take in a key-value pair and store the value at a particular index.

A general outline of how .assign() will work is this:

store the hashed key in a variable arrayIndex

assign the value to the element at arrayIndex in the hash map

**Instructions**

**1.**

Declare a HashMap method called .assign() with the parameters key and value.

Checkpoint 2 Passed

**2.**

Declare a constant called arrayIndex with the value of the hashed and compressed key.

Checkpoint 3 Passed

Hint

Use the HashMap class method that hashes and compresses keys.

**3.**

Assign the value to the element at the index you derived from hashing, arrayIndex.

Checkpoint 4 Passed

Hint

The hash map array can be accessed through the .hashmap property.

**4.**

Check your work. At the bottom of the **HashMap.js** file store a new instance of HashMap with a size of 3 in a constant named employees. Assign employees the key-value pair '34-567' and 'Mara', then log the hash map.

Checkpoint 5 Passed

Hint

The hash map’s array can be accessed through the .hashmap property.

**HashMap.js**

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    this.hashmap[arrayIndex] = value;

  }

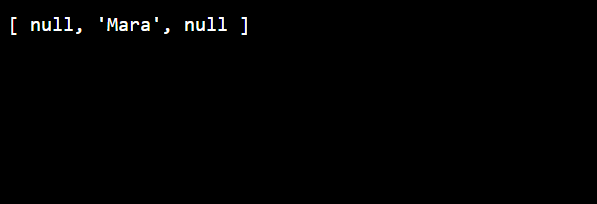
}

module.exports = HashMap;

const employees = new HashMap(3);

employees.assign('34-567', 'Mara');

console.log(employees.hashmap);

****

**Retrieve**

To be a fully functional hash map, we have to be able to retrieve the values we are storing. To implement retrieval for our hash map we’ll create a new HashMap method, .retrieve().

This method will make use of .hash()‘s deterministic nature to find the value we’re looking for in the hash map.

**Instructions**

**1.**

Define a method .retrieve() for HashMap. It should have one parameter, key, the key of the value we want to retrieve.

Checkpoint 2 Passed

**2.**

.retrieve() should calculate the array index in the same way .assign() does and then retrieve the value at that index.

Inside of .retrieve() declare a constant arrayIndex with the value of the hashed key. Use the HashMap method that takes a key and returns an index in the hash map’s array.

Checkpoint 3 Passed

Hint

The HashMap method .hash() takes a key and returns a valid index in the hash map’s array.

**3.**

Return the value stored at arrayIndex.

Checkpoint 4 Passed

Hint

The hash map’s array can be accessed through the .hashmap property.

**4.**

Check your work. At the bottom of the **HashMap.js** file declare a new constant glossary that stores a hashmap with a size of 3.

* Add a new key of: 'semordnilap'
* With a value of: 'Words that form different words when reversed'

Log the result of retrieving 'semordnilap' from your glossary.

**HashMap.js**

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    this.hashmap[arrayIndex] = value;

  }

  retrieve(key) {

    const arrayIndex = this.hash(key);

    return this.hashmap[arrayIndex];

  }

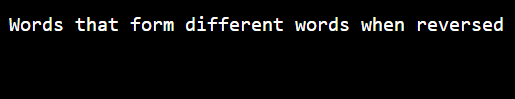
}

module.exports = HashMap;

const glossary = new HashMap(3);

glossary.assign('semordnilap', 'Words that form different words when reversed');

console.log(glossary.retrieve('semordnilap'));

****

**Collisions**

We have a hash map implementation, but what happens when two different keys generate the same index? Run the code in **collision.js** to see a collision in action.

Instead of returning 'marsh plant' and 'forest animal' we retrieve 'forest animal' twice. This is because both key-value pairs are assigned to the same index 0 and the first value, 'marsh plants' was overwritten.

When two different keys resolve to the same array index this is called a collision. In our current implementation, all keys that resolve to the same index are treated as if they are the same key. This is a problem because they will overwrite one another’s values.

**Instructions**

**1.**

Run the code in the text editor to see the result of a collision between two keys.

**collision.js**

const LinkedList = require('./LinkedList');

const Node = require('./Node');

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    this.hashmap[arrayIndex] = value;

  }

  retrieve(key) {

    const arrayIndex = this.hash(key);

    return this.hashmap[arrayIndex];

  }

}

module.exports = HashMap;

const parkInventory = new HashMap(2);

parkInventory.assign('reed', 'marsh plant');

parkInventory.assign('deer', 'forest animal');

console.log(parkInventory.retrieve('reed'));

console.log(parkInventory.retrieve('deer'));

**LinkedList.js**

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output += `<tail>`;

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

    if (currentNode === null) {

      return null;

    } else if (currentNode.data === data) {

      return currentNode;

    } else {

      return this.findNodeRecursively(data, currentNode.next);

    }

  }

}

module.exports = LinkedList;

**Node.js**

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;