07. DATASTRUCTS Hash Tables.

Hashing is a common technique for storing data in such a way that the data can be inserted and retrieved very quickly. Hashing uses a data structure called a **hash table**.

Although hash tables provide fast insertion, deletion, and retrieval, **they perform poorly for operations that involve searching, such as finding the minimum and maximum values in a data set.** For these operations, other data structures such as the binary search tree are more appropriate. We'll learn how to implement a hash table in this chapter and learn when it's appropriate to use hashing as a data storage and retrieval technique.

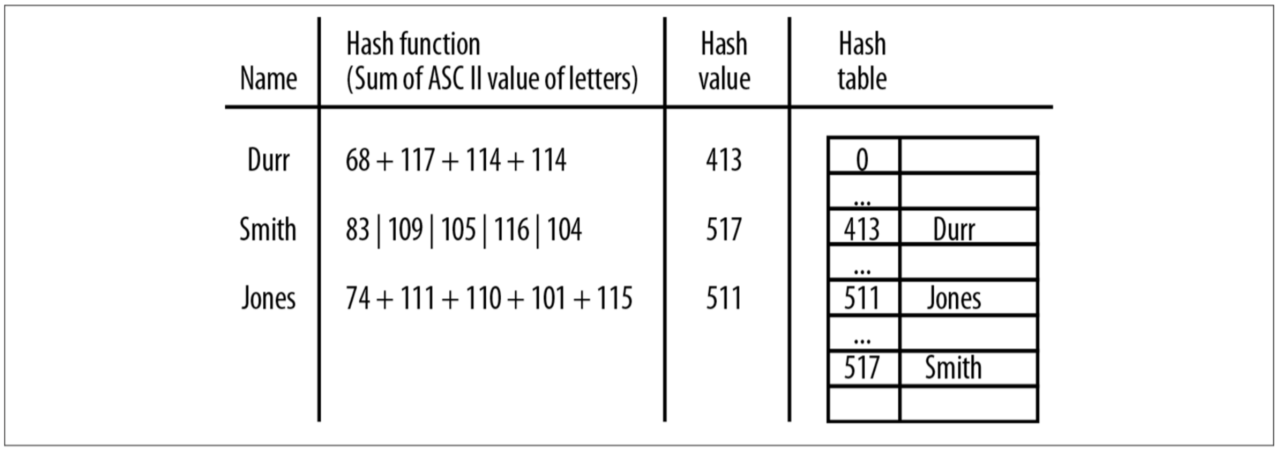
7.1. An Overview of Hashing

The **hash table** data structure is designed around an **array**. The array consists of elements 0 through some predetermined size, though we can increase the size when necessary. **Each data element is stored in the array based on an associated data element called the key,** which is similar to the concept of the key we examined with the dictionary data structure.

To store a piece of data in a hash table, **the key is mapped into a number in the range of 0 through the hash table size, using a hash function.** Ideally, the hash function stores each key in its own array element. However, because there are an unlimited number of possible keys and a limited number of array elements (theoretical in JavaScript), a more realistic goal of **the hash function is to attempt to distribute the keys as evenly as possible among the elements of the array.**

Even with an efficient hash function, **it is possible for two keys to hash (the result of the hash function) to the same value.** This is called a **collision**, and we need a strategy for handling collisions when they occur. We'll discuss how to deal with collisions in detail later in the chapter. The last thing we have to determine when creating a hash function is **how large an array to create for the hash table.** One constraint usually placed on **the array size is that it should be a prime number.** We will explain why this number should be prime when we examine the different hash functions. After that, there are several different strategies for determining the correct array size, all of them based on the technique used to handle collisions, so we will examine this issue when we discuss handling collisions.

Next figure illustrates the concept of hashing using the example of a small phone book.



7.2. A Hash Table Class.

We need a class to represent the **hash table**. The class will include functions for computing hash values, a function for inserting data into the hash table, a function for retrieving data from the hash table, and a function for displaying the distribution of data in the hash table, as well as various utility functions we might need. Here is the constructor function for our ***HashTable*** class:

Example: *HashTable* class constructor:

```
class HashTable {
   constructor(){
      this.table = new Array(137);
   }
}
```

7.2.1 Choosing a Hash Function.

The choice of a hash function depends on the data type of the key. If your key is an integer, then the simplest hash function is to return the key modulo the size of the array. There are circumstances when this function is not recommended, such as when the keys all end in 0 and the array size is 10. This is one reason the array size should always be a prime number, such as 137, which is the value we used in the preceding constructor function. Also, if the keys are random integers, then the hash function should more evenly distribute the keys. This type of hashing is known as **modular hashing**.

In many applications, **the keys are strings**. Choosing a hash function to work with string keys proves to be more difficult and should be chosen carefully. A simple hash function that at first glance seems to work well is to **sum the ASCII value of the letters in the key**. The hash value is then that sum modulo the array size. Here is the definition for this simple hash function:

Example: a Hash Table *simpleHash*() method:

```
simpleHash(data) {
   var total = 0;
   for (var i = 0; i < data.length; ++i) {
      total += data.charCodeAt(i);
   }
   return total % this.table.length;
}
```

We can finish up this first attempt at the *HashTable* class with definitions for **put**() and **showDistro**(), which place the data in the hash table and display the data from the hash table respectively. Here is the complete class definition:

Example: *HashTable* class implementation.

```
class HashTable {
  constructor(){
     this.table = new Array(137);
  }

  put(data) {
     var pos = this.simpleHash(data);
     this.table[pos] = data;
 }

  simpleHash(data) {
     var total = 0;
     for (var i = 0; i < data.length; ++i) {
        total += data.charCodeAt(i);
     }
     return total % this.table.length;
  }
```

```
      showDistro() {
         var n = 0;
         for (var i = 0; i < this.table.length; ++i) {
            if (this.table[i] != undefined) {
               console.log(i + ": " + this.table[i]);
            }
         }
      }

   }
```

The next code demonstrates how the *simpleHash*() function works.

Example: Hashing using a simple hash function.

```
   var  someNames  =  ["David",  "Jennifer",  "Donnie",  "Raymond",
   "Cynthia", "Mike", "Clayton", "Danny", "Jonathan"];

   var hTable = new HashTable();

   for (var i = 0; i < someNames.length; ++i) {
      hTable.put(someNames[i]);
   }

   hTable.showDistro();
```

Output: Here is the output from the previous program.

```
35: Cynthia
45: Clayton
57: Donnie
77: David
95: Danny
116: Mike
132: Jennifer
134: Jonathan
```

The *simpleHash*() function computes a hash value by summing the ASCII value of each name using the JavaScript function *charCodeAt*() to return a character's ASCII value.

- The ***put***() function receives the array index value from the *simpleHash*() function and stores the data element in that position.

- The ***showDistro***() function displays where the names are actually placed into the array using the hash function.

As you can see, the data is not particularly evenly distributed. **The names are bunched up at the beginning and at the end of the array.** There is an even bigger problem than just the uneven distribution of names in the array, however. If you pay close attention to the output, you'll see that not all the names in the original array of names are displayed. Let's investigate further by adding a console.log() statement to the *simpleHash*() function:

Example:

```
simpleHash(data) {
   var total = 0;
   for (var i = 0; i < data.length; ++i) {
      total += data.charCodeAt(i);
   }
   console.log("Hash value: " + data + " -> " + total);
   return total % this.table.length;
}
```

When we run the program again, we see the following output:

Output:

```
Hash value: David -> 488
Hash value: Jennifer -> 817
Hash value: Donnie -> 605
Hash value: Raymond -> 730
Hash value: Cynthia -> 720
Hash value: Mike -> 390
Hash value: Clayton -> 730
Hash value: Danny -> 506
Hash value: Jonathan -> 819

35: Cynthia
45: Clayton
57: Donnie
77: David
95: Danny
116: Mike
132: Jennifer
134: Jonathan
```

7.2.3. A Better Hash Function

To avoid collisions, you first need to make sure the array you are using for the hash table is sized to a prime number. This is necessary due to the use of modular arithmetic in computing the key. The size of the array needs to be greater than 100 in order to more evenly disperse the keys in the table. Through experimentation, we found that the first prime number greater than 100 that didn't cause collisions for the data set used in the previous program is 137. When smaller prime numbers close to 100 were used, there were still collisions in the data set.

After properly sizing the hash table, the next step to avoiding hashing collisions is to compute a better hash value. An algorithm known as **Horner's method** does the trick. Without getting too deep into the mathematics of the algorithm, our new hash function still works by **summing the ASCII values of the characters of a string, but it adds a step by multiplying the resulting total by a prime constant.** Most algorithm textbooks suggest a small prime number, such as 31, but for our set of names 31 didn't work; however, 37 worked without causing collisions. We now present a new, better hash function utilizing Horner's method:

Example: a Hash Table *betterHash*() method:

```
betterHash(string, arr) {
   const H = 37;
   var total = 0;
   for (var i = 0; i < string.length; ++i) {
      total += H * total + string.charCodeAt(i);
   }
   total = total % arr.length;
   return parseInt(total);
}
```

7.2.4. Hashing Integer Keys.

In the last section we worked with string keys. In this section, we introduce **how to hash integer keys**. The data set we're working with is student grades. **The key is a nine-digit student identification number,** which we will generate randomly, along with the student's grade.

Example: Here are the functions we use to generate the student data (ID and grade):

```
function getRandomInt (min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

function genStuData(arr) {
   for (var i = 0; i < arr.length; ++i) {
      var num = "";
      for (var j = 1; j <= 9; ++j) {
         num += Math.floor(Math.random() * 10);
      }
      num += getRandomInt(50, 100);
      arr[i] = num;
   }
 }
```

The ***getRandomInt***() function allows us to specify a maximum and minimum random number. For a set of student grades, it is reasonable to say that the minimum grade is 50 and the maximum grade is 100.

The ***getStuData***() function generates student data. The inner loop generates the student ID number, and right after the inner loop finishes, a random grade is generated and concatenated to the student ID. Our main program will separate the ID from the grade. The hash function will total the individual digits in the student ID to compute a hash value using the *simpleHash*() function.

Example:

```
var numStudents = 10;
var arrSize = 97;
var idLen = 9;
var students = new Array(numStudents);

genStuData(students);
console.log("Student data: \n");
for (var i = 0; i < students.length; ++i) {
          console.log(students[i].substring(0,8)    +    "    "    +
               students[i].substring(9));
}
console.log("\n\nData distribution: \n");

var hTable = new HashTable();
for (var i = 0; i < students.length; ++i) {
   hTable.put(students[i]);
}
hTable.showDistro();
```

When we run the program again, we see the following output:

Output:

```
Student data:
24553918 70
08930891 70
41819658 84
04591864 82
75760587 91
78918058 87
69774357 53
52158607 59
60644757 81
60134879 58

Data distribution:
41: 52158607059
42: 08930891470
47: 60644757681
50: 41819658384
53: 60134879958
54: 75760587691
61: 78918058787
```

Once again, our hash function creates a collision, and not all of the data is stored in the array. Actually, if you run the program several times, all of the data will sometimes get stored, but the results are far from consistent. We can play around with array sizes to see if we can fix the problem, or we can simply change the hash function called by the put() function and use ***betterHash***().

Output: When using *betterHash*() with the student data, we get the following output:

```
Student data:
74980904 65
26410578 93
37332360 87
86396738 65
16122144 78
75137165 88
70987506 96
04268092 84
95220332 86
55107563 68

Data distribution:
10: 75137165888
```

```
      34: 95220332486
      47: 70987506996
      50: 74980904265
      51: 86396738665
      53: 55107563768
      67: 04268092284
      81: 37332360187
      82: 16122144378
      85: 26410578393
```

The lesson here is obvious: *betterHash*() is the superior hashing function for strings and for integers.

7.3. Storing and Retrieving Data in a Hash Table.

Now that we've covered hash functions, we can apply this knowledge to use a hash table to actually store data. To do this, we have to modify the *put*() function so that it accepts both a key and data, hashes the key, and then uses that information to store the data in the table. Here is the definition of the new *put*() function:

Example: *put*() method.

```
put(key, data) {
  var pos = this.betterHash(key);
  this.table[pos] = data;
}
```

The *put*() function hashes the key and then stores the data in the position of the table computed by the hash function.

Next we need to define the **get**() function so that we can retrieve data stored in a hash table. This function must, again, hash the key so that it can determine where the data is stored, and then retrieve the data from its position in the table. Here is the definition:

Example: *get*() method.

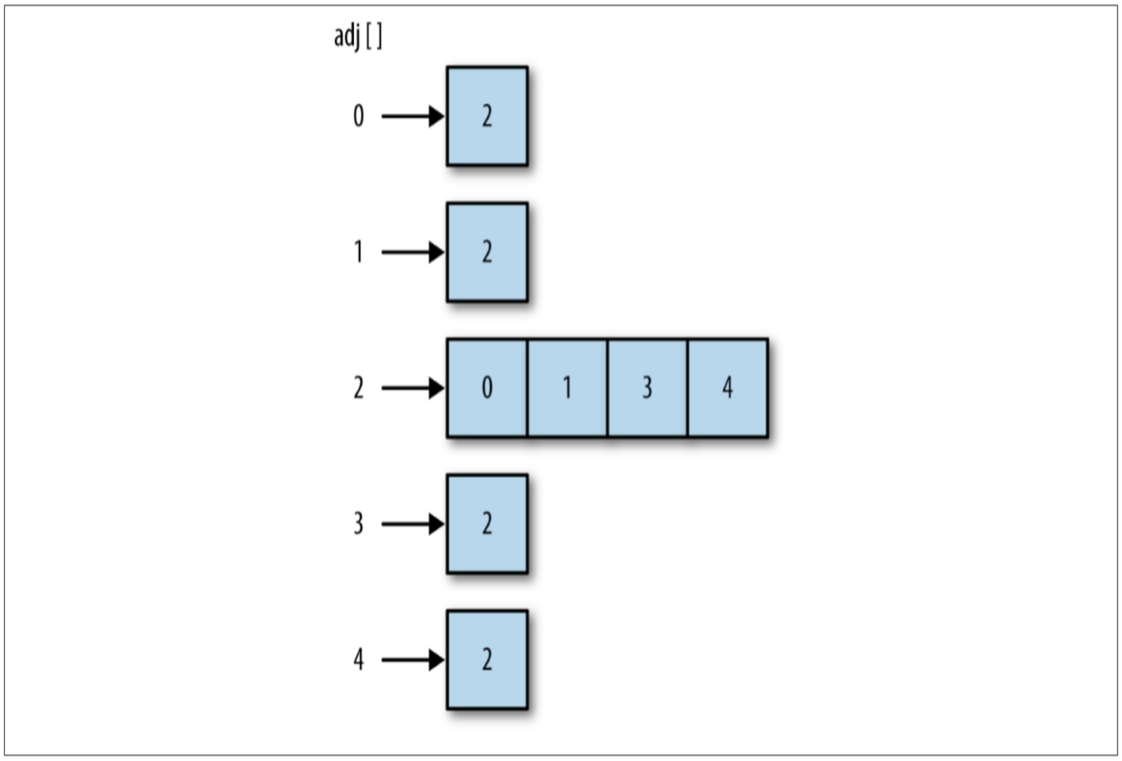
```
get(key) {
   return this.table[this.betterHash(key)];
}
```

7.4. Handling Collisions

A collision occurs when a hash function generates the same key for two or more values. The second part of a hash algorithm involves resolving collisions so that all keys are stored in the hash table. In this section, we look at two means of collision resolution: **separate chaining** and **linear probing**.

7.4.1. Separate Chaining

When a collision occurs, we still need to be able to store the key at the generated index, but **it is physically impossible to store more than one piece of data in an array element.** Separate chaining is a technique where **each array element of a hash table stores another data structure**, such as another array, which is then used to store keys. Using this technique, if two keys generate the same hash value, each key can be stored in a different position of the secondary array. Next figure illustrates how separate chaining works.



To implement separate chaining, **after we create the array to store the hashed keys, we call a function that assigns an empty array to each array element of the hash table.** This creates a two-dimensional array. The following code defines a function, ***buildChains***(), to create the second array, as well as a small program that demonstrates how to use *buildChains*():

Example:

```
function buildChains() {
   for (var i = 0; i < this.table.length; ++i) {
      this.table[i] = new Array();
   }
}
```

Add the preceding code, along with a declaration of the function, to the definition of the *HashTable* class. A program to test separate chaining is shown next.

Example: Using separate chaining to avoid collisions.

```
var hTable = new HashTable();

hTable.buildChains();

var  someNames  =  ["David",  "Jennifer",  "Donnie",  "Raymond",
"Cynthia", "Mike", "Clayton", "Danny", "Jonathan"];

for (var i = 0; i < someNames.length; ++i) {
   hTable.put(someNames[i]);
}

hTable.showDistro();
```

In order to properly display the distribution after hashing with separate chaining, we need to modify the *showDistro*() function in the following way to **recognize that the hash table is now a multidimensional array:**

Example:

```
showDistro() {
   var n = 0;
   for (var i = 0; i < this.table.length; ++i) {
      if (this.table[i][0] != undefined) {
         console.log(i + ": " + this.table[i]);
      }
   }
}
```

When we run the program, we get the following output:

Output:

```
60: David
68: Jennifer
69: Mike
70: Donnie,Jonathan
78: Cynthia,Danny
88: Raymond,Clayton
```

Next we need to define the **put**() and **get**() functions that will work with separate chaining. The put() function hashes the key and then attempts to store the data in the first cell of the chain at the hashed position. If that cell already has data in it, the function searches for the first open cell and stores the data in that cell.

Example: Here is the code for the *put*() function:

```
put(key, data) {
   var pos = this.betterHash(key);
   var index = 0;
   if (this.table[pos][index] == undefined) {
      this.table[pos][index+1] = data;
   }
   else {
      while (this.table[pos][index] != undefined) {
         ++index;
      }
      this.table[pos][index] = data;
   }
 }
```

Unlike the example earlier when we were just storing keys, this *put*() function has to store both keys and values. The function uses a pair of the chain's cells to store a **key-value pair**; the first cell stores the key and the adjacent cell of the chain stores the value.

The ***get***() function starts out by hashing the key to get the position of the key in the hash table. Then the function searches the cells until it finds the key it is looking for. When it finds the correct key, it returns the data from the adjacent cell to the key's cell. If the key is not found, the function returns undefined. Here is the code:

Example: Here is the code for the *get*() function:

```
get(key) {
       var index = 0;
       var pos = this.simpleHash(key);
       if (this.table[pos][index] == key) {
          return this.table[pos][index];
       }
       else {
          while ((this.table[pos][index] != undefined) &&
                 (this.table[pos][index] != key)) {
             index++;
          }
          return this.table[pos][index];
       }
       return undefined;
 }
```

7.4.2. Linear Probing.

A second technique for handling collisions is called **linear probing**. Linear probing is an example of a more general hashing technique called **open-addressing hashing**. With linear probing, **when there is a collision, the program simply looks to see if the next element of the hash table is empty. If so, the key is placed in that element. If the element is not empty, the program continues to search for an empty hash-table element until one is found.** This technique makes use of the fact that any hash table is going to have many empty elements and it makes sense to use the space to store keys.

Linear probing should be chosen over separate chaining when your array for storing data can be fairly large. Here's a formula commonly used to determine which collision method to use: if the size of the array can be up to half the number of elements to be stored, you should use separate chaining; but if the size of the array can be twice the size of the number of elements to be stored, you should use linear probing.

To demonstrate how linear probing works, we can rewrite the *put*() and *get*() functions to work with linear probing. In order to create a realistic data-retrieval system, we have to modify the *HashTable* class by adding a second array to store values. The table array and the values array work in parallel, so that when we store a key in a position in the tables array, we store a value in the corresponding position in the values array. Add the following code to the *HashTable* constructor:

```
this.values = [];
```

Example: Here is the code for the *put*() function:

```
put(key, data) {
   var pos = this.betterHash(key);
   if (this.table[pos] == undefined) {
      this.table[pos] = key;
      this.values[pos] = data;
   }
   else {
      while (this.table[pos] != undefined) {
         pos++;
      }
      this.table[pos] = key;
      this.values[pos] = data;
```

```
         }
      }
```

The code for the *get*() function begins searching the hash table at the hashed position of the key. If the data passed to the function matches the key found at that position, the corresponding data in the values position is returned. If the keys don't match, the function loops through the hash table until it either finds the key or reaches a cell that is undefined, meaning the key was never placed into the hash table. Here's the code:

Example: Here is the code for the *get*() function:

```
       get(key) {
          var hash = -1;
          hash = this.betterHash(key);
          if (hash > -1) {
             for (var i = hash; this.table[hash] != undefined; i++) {
                if (this.table[hash] == key) {
                   return this.values[hash];
                }
             }
          }
          return undefined;
        }
```