# Hashing

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 discover when it is appropriate to use hashing as a data storage and retrieval technique.

## 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 some piece of data called the *key*. To store an element in a hash table, the key is mapped into a number in the range of 0 through the hash table size, using a function called 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 (theoretical in JavaScript) limited number of array elements, 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 have to have a strategy for dealing with 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 to create the array used in the hash table. One constraint usually placed on the array size is that the size be a prime number. We will explain why when we examine the different hash functions. After that, there are several different strategies for determining the proper array size, all of them based on the technique used to handle collisions, so we'll examine this issue when we discuss handling collisions.

## A Hash Table Class

We need a class to represent a hash table. The methods that will make up the class include methods for computing hash values, a method for inserting data into the hash table, a method for retrieving data from the hash table, a method for displaying the distribution of data in the hash table, as well as various utility methods we might need.

Here is the constructor function for our hash table class, which lists some of the methods we will be developing in the sections below:

function HashTable() {

this.table = new Array(137);

this.simpleHash = simpleHash;

this.betterHash = betterHash;

this.showDistro = showDistro;

this.put = put;

//this.get = get;

}

The get() method is commented out for now until we describe its definition later in the chapter.

## Choosing a Hash Function

The choice of a hash function depends on the data type of the key you are using. 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 method is not recommended, such as when the keys all end in zero and the array size is 10. This is one reason why the array size should always be a prime number. 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, however, the keys are strings. Choosing a hash function to work with string keys proves to be more difficult and the hash function 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. The following program demonstrates how this function works (including the put() method for inserting data into the hash table and the showDistro() method for displaying the contents of the hash table):

function simpleHash(data) {

var total = 0;

for (var i = 0; i < data.length; ++i) {

total += data.charCodeAt(i);

}

return total % this.table.length;

}

function put(data) {

var pos = this.simpleHash(data);

this.table[pos] = data;

}

function showDistro() {

var n = 0;

for (var i = 0; i < this.table.length; ++i) {

if (this.table[i] != undefined) {

print(i + ": " + this.table[i]);

}

}

}

// main program

var someNames = ["David", "Jennifer", "Donnie", "Raymond",

"Cynthia", "Mike", "Clayton", "Danny", "Bryan"];

var hTable = new HashTable();

for (var i = 0; i < someNames.length; ++i) {

hTable.put(someNames[i]);

}

hTable.showDistro();

The output from this program is:

35: Cynthia

45: Clayton

57: Donnie

77: David

95: Danny

97: Bryan

116: Mike

132: Jennifer

The hash function computes a hash value by totaling the ASCII value of each name using the JavaScript function charCodeAt() to return a character's ASCII value. The put() method simply receives the array index value from the simpleHash() function and stores the data element in that position. The showDistro() function shows us where the names are actually placed into the array by the hash function. As you can see, the distribution is not particularly even. The names are bunched together at the beginning of the array and at the end.

There is an even bigger problem than the 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 print() statement to our hash function:

function simpleHash(data) {

var total = 0;

for (var i = 0; i < data.length; ++i) {

total += data.charCodeAt(i);

}

print("Hash value: " + data + " -> " + total);

return total % this.table.length;

}

Now we rerun the program and see the following 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: Bryan -> 508

35: Cynthia

45: Clayton

57: Donnie

77: David

95: Danny

97: Bryan

116: Mike

132: Jennifer

Now the problem is apparent – the strings "Clayton" and "Raymond" hash to the same value, causing a collision. Because of the collision, only "Clayton" is stored in the array. We can improve our hash function to avoid such collisions, as discussed in the next section.

## A Better Hash Function

The solution hash value collisions is two-pronged. First, we need to change the size of the array to a prime number. If we set the size of the array to a prime number less than 100, we don't resolve the problem. If we set the size of the array to the first prime number larger than 100, we still don't resolve the problem. The first prime number larger than 100 that doesn't result in a collision is 107, so that is the size of the array.

The next step is to compute a better hash value. An algorithm known as *Horner's method* does the trick. Without getting into the mathematics of the method, the hash function works by still 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 algorithms textbooks suggest a small prime number, such as 31, but for this particular application, 31 didn't work, but 37 worked fine.

Here is a new hash function utilizing Horner's method:

function 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;

if (total < 0) {

total += arr.length-1;

}

return parseInt(total);

}

You'll notice that the function also includes a check to make sure that the resulting value of the function is at least equal to 0, setting the total to the length of the array minus 1 if the total is less than 0.

Here is the output when we plug this new function into our program:

17: Cynthia

25: Donnie

30: Mike

33: Jennifer

57: Clayton

65: David

66: Danny

88: Bryan

99: Raymond

## Hashing Integer Keys

In the last section, we worked with keys that were strings. Let's see how hashing works when we are working with integer keys. The data we are storing in an array are student grades. The keys are 9 digit student identification numbers, which we will generate randomly, along with the grades. Here are the functions we use to generate 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 += Math.floor(Math.random() \* 101);

arr[i] = num;

}

}

We've introduced a new function, getRandomInt(), so that we can specify a minimum and maximum value for a random number. For a set of grades, it's reasonable to say that the minimum grade will be 50 and, of course, 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 split the grade from the id. The hash function will simply total the individual digits in the student id to compute a hash value.

Our program will use the simpleHash() function to generate hash values. Here is the main program:

var numStudents = 10;

var arrSize = 97;

var idLen = 9;

var students = new Array(numStudents);

genStuData(students);

var grades = new Array(arrSize);

for (var i = 0; i < students.length; ++i) {

var student = students[i].substr(0,idLen);

var grade = parseInt(students[i].substr(idLen+1,2));

if (grade < 30) { grade += 50; }

print(student, grade);

var hashVal = simpleHash(student, grades);

grades[hashVal] = grade;

}

showDistro(grades);

Here is the output from this program:

Student data:

931183365 61

504954315 96

286788858 62

228135408 75

744510987 61

117333091 10

825460290 86

440040562 53

638448459 89

720751614 62

Data distribution

7: 62

69: 53

72: 10

77: 62

80: 86

83: 61

89: 61

95: 89

Once again, our hash function creates a collision and not all of the data is stored in the array. Interestingly, sometimes the program will store all the data but the results are not consistent. We can play around with array sizes to see if we can fix the problem or we can simply change our hash function and use betterHash(), which utilizes Horner's method. With this function, we get the following output:

Student data:

416377635 61

772057220 56

651428733 76

885443949 64

703629868 91

020992156 60

218724257 55

427457242 57

430797907 63

737939905 81

Data distribution

8: 56

11: 64

12: 76

14: 81

18: 57

33: 91

56: 60

59: 61

68: 63

82: 55

The lesson here is obvious – the betterHash() function is the better-performing hash function.

## Handling Collisions

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

### Separate Chaining

When a collision occurs, we still need to be able to store the key at that index but it is physically impossible to store more than one piece of data in an array element. With separate chaining, each array element stores another data structure, such as an array, which is then used to store keys. This way, if two keys generate the same hash value, each key can be stored in a different position of the list. Figure x.1 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 hash array element. We use a simple function named buildChains() to create the second array (we'll also refer to this array as a chain). Here is the code for the buildChains() function and a code fragment that demonstrates how the buildChains() function works:

function buildChains(arr) {

for (var i = 0; i < arr.length; ++i) {

arr[i] = new Array();

}

}

var names = new Array(97);

buildChains(names);

for (var i = 0; i < someNames.length; ++i) {

var name = someNames[i];

var hashVal = simpleHash(name, names);

names[hashVal].push(name);

}

### Linear Probing

A second technique for handling collisions is called *linear probing*. Linear probing is an example of a more general 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 this space to store keys.

## Searching Hash Table Data Stored with Separate Chaining

To search for a key stored in a hash table, we need to compute the hash value for the specified key. Here is a function to accomplish the task:

function inHash(key, arr) {

var hash = simpleHash(key, arr);

var n = 0;

if (key == arr[hash][n]) {

return true;

}

else {

while (arr[hash][n] != undefined) {

if (arr[hash][n] == key) {

return true;

}

++n;

}

}

return false;

}

The function first computes the hash value of the key. Then the first array element in the chain at the hashed index position is checked to see if it is equal to the key. If so, the function returns true. If not, the remaining elements of the chain are checked, stopping if the current chain element is undefined, which means the end of chain has been reached and the key is not stored in the hash table.

Here is the complete program with the inHash() function:

var someNames = ["David", "Jennifer", "Donnie", "Raymond",

"Cynthia", "Mike", "Clayton", "Danny", "Bryan"];

var names = new Array(97);

buildChains(names);

for (var i = 0; i < someNames.length; ++i) {

var name = someNames[i];

var hashVal = simpleHash(name, names);

names[hashVal].push(name);

}

showDistro(names);

for (var i = 1; i < 4; ++i) {

putstr("Search for? ");

var name = readline();

if (inHash(name, names)) {

print("Found " + name);

}

else {

print("Can't find " + name);

}

}

Here is the output from running this program:

Mike

David

Danny

Donnie

Bryan

Jennifer

Cynthia

Raymond

Clayton

Search for? Raymond

Found Raymond

Search for? Cynthia

Found Cynthia

Search for? Alisa

Can't find Alisa

## Exercises

1. Write a function, get(), that returns the value associated with a key in a hash table using separate chaining.
2. Rewrite the get() function from Exercise 1 for a hash table using linear probing.