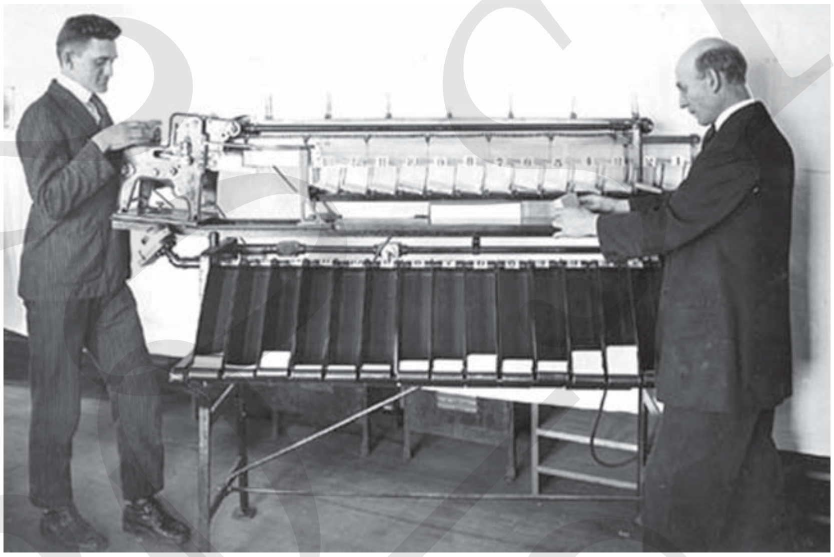
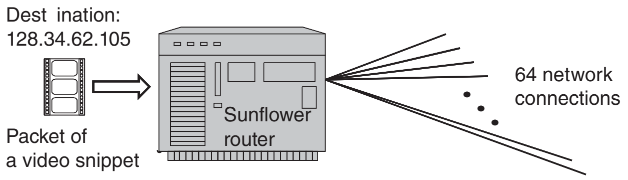
Read first 3 sections carefully; you can skip cuckoo hashing and universal hashing.

Chapter 6 Hash Tables


Operating a card sorter, 1920. U.S. Census Bureau.

Contents



Suppose you are working on a development team that is designing the next generation of a network router for a major technology company. Such devices processes information packets, allowing them to move through networks having lots of interconnections. In this case, your team's job is to design a router, code named "Sunflower," that can process 64 streams of high-definition video at a time.

When a packet is received from one of the 64 cables that connect to the Sunflower router, it must examine destination information stored in the beginning of that packet and quickly decide along which of the 63 remaining cables to send it. Indeed, to avoid introducing annoying pauses in a video stream, the Sunflower router needs to process each such packet in at most 25 microseconds. Your job is to write the software that processes the destination information for deciding where to send each packet.

Viewed abstractly, each packet can be modeled as a pair, (k, x), where k is a key indicating the destination for this packet and x is the data contained in this packet, which would, in this case, be a snippet of some video stream. To process such a packet, your software needs to maintain a collection of pairs, (k, c), each of which indicates the cable, c, where the Sunflower router should send a packet with the destination k. Your system must support an operation, put(k, c), which adds a key-cable pair to the collection, and an operation, get(k), which returns the cable number in the collection for a given destination key, k.

One possibility, of course, is to use a linked list to store (k, c) pairs. This implementation choice would allow you to perform the put(k, c) operation in $O(1)$ time, since you could simply put each new pair at the beginning of the list. But you have correctly realized that such a solution would take $O(n)$ time to process a single get(k) operation on a collection of n pairs, since you would, in general, have to search through the entire list of n pairs to find a pair with the key k. Therefore, such a solution would put your team significantly over the 25 microsecond time limit if n is relatively large. Fortunately, there is a better choice, which is to use an instance of the ***hash table*** data structure we discuss in this chapter.

This data structure is able to achieve $O(1)$ expected time for both get and put operations. Indeed, we describe variations of this data structure that can achieve worst-case $O(1)$ time performance for either get or put operations, with the other operation still running in $O(1)$ expected time. Because of such performance bounds, hash tables are used in a host of different applications besides network routers, including operating systems, computer games, and bioinformatics.

6.1 Maps

The main idea at the heart of the hash table data structure is that it allows users to assign keys to elements and then use those keys later to look up or remove elements. (See Figure 6.1.) This functionality defines a data structure known as a ***dictionary*** or ***map***. That is, this structure supports methods for the insertion, removal, and searching of values in terms of keys associated with those values.

Figure 6.1: A conceptual illustration of a map data structure. Keys are like labels assigned to file folders, which serve as the values. The resulting item (labeled file folders) is then inserted into the map (file cabinet) by a file clerk. The keys can be used later to retrieve or remove the items. (Included image: LC-DIG-ppmsca-03084, 1945. U.S. government image, U.S. Office of War Information.)

6.1.1 The Map Definition

A map stores a collection of key-value pairs, (k, v), which we call ***items***, where k is a key and v is a value that is associated with that key. For example, in a map storing student records (such as the student's name, address, and course grades), the key might be the student's ID number. That is, a ***key*** is an identifier that is assigned by an application or user to an associated value, which we allow to be any data element.

For the sake of generality, some definitions allow a map to associated multiple values with the same key. Nevertheless, in most applications we would probably want to disallow items with the same key (for example, in a map storing student records, we would probably want to disallow two students having the same ID). In such cases when keys are unique, then the key associated with an object can be viewed as an "address" for that object in memory. Indeed, such maps are sometimes referred to as "associative stores," because the key associated with an object determines its "location" in the map. Thus, we assume here that keys are unique and we refer to a map that allows for multiple values for the same key as a ***multimap***.

As ***map*** data structure, M, supports the following fundamental methods:

get(k): If M contains an item with key equal to k, then return the value of such an item; else return a special element NULL.

put(k, v): Insert an item with key k and value v; if there is already an item with key k, then replace its value with v.

remove(k): Remove from M an item with key equal to k, and return this item. If M has no such item, then return the special element NULL.

Note that when operations get(k) and remove(k) are unsuccessful (that is, the map M has no item with key equal to k), we use the convention of returning a special element NULL. Such a special element is known as a ***sentinel***. Alternatively, we could have had these methods indicate an error or exception in such cases, but it would not normally be considered an error to search for a key that happens not to be present in a map.

In addition, a map can implement other supporting methods, such as size() and isEmpty() methods for containers. Moreover, we could include methods for listing out the items, values, or keys in M. Still, the above three methods are the essential ones for a Map structure.

As mentioned above, the keys associated with values in a map are often meant to be "addresses" for those values. In addition to their applications in Internet routers, as in the Sunflower router mentioned above, another application of a map is in a compiler's symbol table, where each name in a program is a key associated with a variable, function, or class. In this case, we would store name-value associations, where each name serves as the "address" for properties about a variable's type and value.

6.1.2 Lookup Tables

In some cases, the universe of keys that will be used for a map is the set of integers in the range $[0, N-1]$, for a reasonably small value of N. In such a scenario, there is an almost trivial way of implementing a map—namely, we can allocate an array, A, of size N, where each cell of A is thought of as a "bucket" that can hold a single key-element pairs (or a pointer to such a pair). Since we assume that keys are distinct, we can use A to store any key-value pair (k, v) by placing that pair in the cell $A[k]$. In this case, we refer to this implementation as a ***lookup table***, since the key k allows us to simply "look up" the item for k by a single array-indexing operation.

Performing the essential operations for a map in this case is quite simple. We begin by allocating the array A so that every cell is initially associated with the special NULL object. Then, we perform the map operations as follows:

- To perform a put(k, v) operation, we assign (k, v) to $A[k]$.
- To perform a get(k) operation, we return $A[k]$.
- To perform a remove(k) operation, we return $A[k]$ and then we assign the NULL item to $A[k]$.

(See Figure 6.2.)

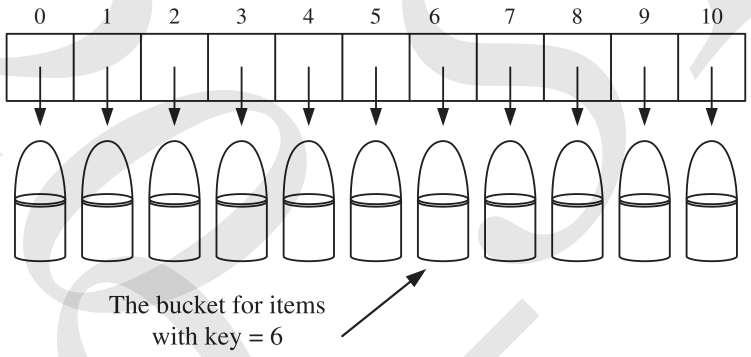


Figure 6.2: How a lookup table works.

Analysis of a Lookup Table

The analysis of the performance of a lookup table is also simple. Each of the essential map methods runs in $O(1)$ time in worst case. Thus, in terms for time performance, the lookup table is optimal.

There are some drawbacks with this implementation, however. In terms of space usage, we note that the total amount of memory used for a lookup table is $\Theta(N)$. In this case, we refer to the size, N, of the array A as being the ***capacity*** of this map implementation. Such an amount of space is certainly reasonable if the number of items, n, being stored in the map is close to N. But if N is large relative to n, then one drawback of this implementation is that it is wasteful of space. Another drawback with this implementation is that it requires keys be unique integers in the range $[0, N-1]$, which is often not the case. Thus, it would be nice to have a mechanism to overcome these drawbacks while still achieving simple and fast methods for implementing the essential operations for a map.

6.2 Hash Functions

If we cannot assume that keys are unique integers in the range $[0, N-1]$, but we nevertheless still want to use something akin to a lookup table for storing key-value pairs, then we need a good way of assigning keys to integers in this range. That is, we need a function, h, called a ***hash function***, that maps each key k in our map to an integer in the range $[0, N-1]$, where N is the capacity of the underlying array for this table. The use of such a function allows us to treat objects, such as strings, as numbers.

Using a Hash Function with a Lookup Table

Equipped with such a function, h, we can apply the lookup table approach to arbitrary keys. The main idea of this approach is to use the hash value, $h(k)$, as an index into our array, A, instead of the key k itself. That is, the idea is to try to store the item (k, v) in the bucket $A[h(k)]$. (See Figure 6.3.)

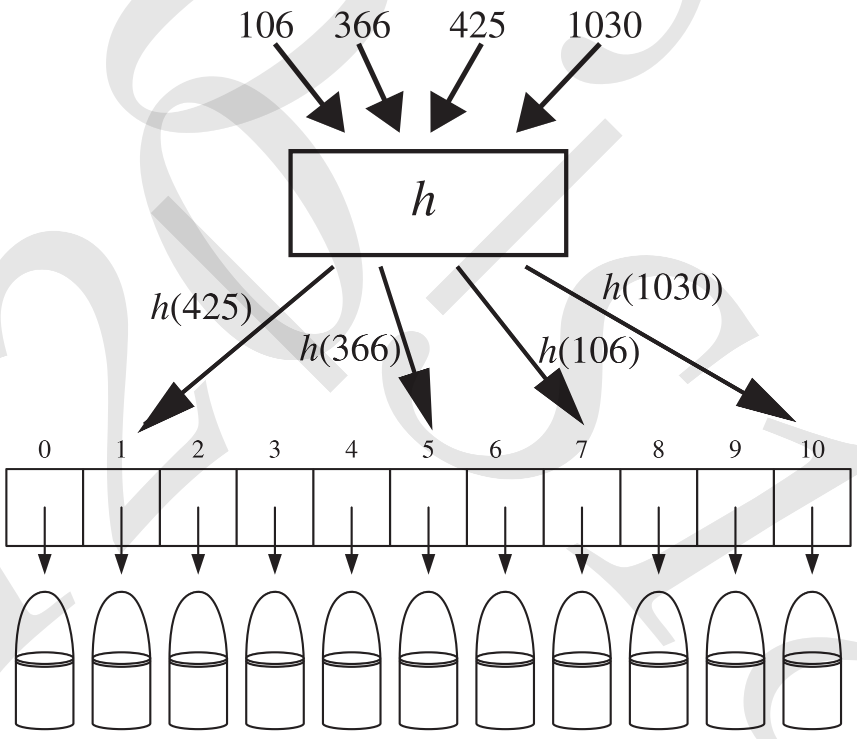


Figure 6.3: How a hash function, h, works with a lookup table. For simplicity, we only show keys, not values.

Collisions

One complication that we have to deal with when using this approach, however, is that it is entirely possible that the hash function, h, could map two distinct keys to the same location in A. That is, we could have $h(k) = h(l)$, for $k \neq l$, for two keys k and l in our map. We refer to such a pair of keys with the same hash value, j, as causing a ***collision*** at j. Likewise, we say that a hash function is "good" if it maps the keys in our map so as to minimize collisions as much as possible. In other words, we would like the hash function h to look as random as possible, since a truly random function would automatically minimize the expected number of collisions for any given hash value, j. For practical reasons, we also would like the evaluation of a given hash function to be fast and easy to compute.

Viewing Keys as Bit Strings, Tuples, or Integers

Any kind of key that is of interest in a computing application must be representable as a binary string. Thus, without loss of generality, we can assume that any key used with a map data structure is a binary string. Of course, any such binary string can be interpreted as belonging to any one of a large number of different data types. Nevertheless, the standard convention for hash functions is to view keys in one of two ways:

- Each key, k, is a tuple of integers, $(x_1, x_2, \ldots, x_d)$, with each x_i being an integer in the range $[0, M-1]$, for some M and d.
- Each key, k, is a nonnegative integer, which could possibly be very large.

For example, if our keys are character strings, such as in the case of the variable names in a programming language or words taken from web pages, then it is probably most natural to view such keys as tuples. Alternatively, if our keys are fixed-length IP addresses of destination domains on the Internet, then it is probably most natural to view them as single integers.

Depending on which of these two viewpoints we take, there are a number of different kinds of hash functions that we can consider. In fact, there are a large number of different hash functions that have been published over the years, each of which is based on the goal of minimizing collisions. Rather than survey all the different existing hash functions, however, we restrict our discussion in this section to some important representative hash functions. This collection is representative of existing hash functions in that most of the existing functions contain elements from the ones we discuss and assume that keys come in one of the above two forms. In some cases, a hashing scheme might use both forms of keys, for instance, first viewing an input key as a tuple and mapping that to some integer based on one hash function, and then further mapping that single integer to some other value based on yet another hash function.

6.2.1 Summing Components

In the case when each of our keys, k, is a d-tuple, of the form

$$(x_1, x_2, \ldots, x_d),$$

where each x_i is an integer, one possible hash function we could use is to sum up the different components in each key. That is, we could compute $h(k)$ as

$$h(k) = \sum_{i=1}^{d} x_i,$$

where this sum is taken over either the integers or is done so that every addition is modulo some integer, p, so that the result is in the range $[0, p-1]$.

A slight variation on this theme is to compute an exclusive-or of all the components of a key, which could be written mathematically as

$$h(k) = \oplus_{i=1}^{d} x_i,$$

where $\oplus$ denotes the bitwise exclusive-or (XOR) operation. Using an XOR is sometimes preferred over addition in that there are no extra complications regarding carry bits when one is doing XORs and the XOR operation itself is often a fast built-in operation for most CPUs. But there are some caveats, as the XOR of a number and itself is always 0, so care should be taken if there are duplicate components in a tuple.

How Symmetry Can Cause Collisions

Unfortunately, such hash functions, which consist of either summing or XORing the components of each key, are actually not that good in most applications. In particular, such a function is fairly poor at avoiding collisions for the case when keys are character strings or other multiple-length objects that can be viewed as tuples of the form $(x_1, x_2, \ldots, x_k)$, as used above, where the order of the x_i's is significant.

For example, consider such a hash function for a character string s that sums the ASCII (or Unicode) values of the characters in s. This hash function produces lots of unwanted collisions for common groups of strings. For instance, `"temp01"` and `"temp10"` collide using this function, as do the four words,

`"stop"`, `"tops"`, `"pots"`, and `"spot"`.

A better hash code in such cases should somehow take into consideration the positions of the x_i's. That is, each index, i, should play a role in the hash function in some way.

6.2.2 Polynomial-Evaluation Functions

An alternative hash code, which does a better job at factoring in the positional information of the components in a key $k = (x_1, x_2, \ldots, x_d)$, is to choose a nonzero constant, $a \neq 1$, and use as a hash function the following:

$$h(k) = x_1 a^{d-1} + x_2 a^{d-2} + \cdots + x_{d-1} a + x_d,$$

which, by Horner's rule (see Exercise C-1.14), can be rewritten as

$$h(k) = x_d + a(x_{d-1} + a(x_{d-2} + \cdots + a(x_3 + a(x_2 + a x_1)) \cdots)).$$

Note that we can evaluate such a hash function in a simple for-loop, which has $d - 1$ iterations; hence, this function can be evaluated using $d - 1$ additions and $d - 1$ multiplications. Mathematically speaking, the hash function, h, in this case is a $(d - 1)$-degree polynomial, which is being evaluated for the argument a, and has the components $(x_1, x_2, \ldots, x_d)$ as its coefficients. We therefore refer to this hash function as a ***polynomial-evaluation*** hash function.

Intuitively, a polynomial-evaluation hash function uses multiplication by the constant a as a way of "making room" for each component in a tuple of values while also preserving a characterization of the previous components. Of course, on a typical computer, evaluating a polynomial will be done using the finite bit representation for a hash code; hence, the value will periodically overflow the bits used for an integer. Since we are more interested in a good spread of the object x with respect to other keys, we simply ignore such overflows or we assume that all the arithmetic is being done modulo some prime number, p. In any case, one of the critical design decisions in using this hash function is choosing the constant, a. Intuitively, a should have some higher-order bits to "shift" over the running partial sum to make room for the new term while also having some lower-order bits to allow each new term to be factored in.

Special Values for English Words

We have done some experimental studies that suggest that 33, 37, 39, and 41 are good choices for a when working with character strings that are English words. In fact, in a list of over 50,000 English words formed as the union of the word lists provided in two variants of Unix, we found that taking a to be 33, 37, 39, or 41 produced less than 7 collisions in each case. It should come as no surprise, then, to learn that actual character string hash functions often choose the polynomial hash function using one of these constants for a. For the sake of speed, however, some implementations only apply the polynomial hash function to a fraction of the characters in long strings, say, every eight characters.

6.2.3 Tabulation-Based Hashing

For the case when each key can be viewed as a tuple, $k = (x_1, x_2, \ldots, x_d)$, for a fixed d, there is another class of hash functions we can use, which involve simple table lookups. These are known as ***tabulation-based hash functions***, and they can be applied even for hashing on small devices, since they make no use of addition or multiplication.

So, let us assume that all our keys are of the form $k = (x_1, x_2, \ldots, x_d)$, for a fixed d, and each x_i is in the range $[0, M-1]$. Then we can initialize d tables, $T_1, T_2, \ldots, T_d$, of size M each, so that each $T_i[j]$ is a uniformly chosen independent random number in the range $[0, N-1]$. We then can compute the hash function, $h(k)$, as

$$h(k) = T_1[x_1] \oplus T_2[x_2] \oplus \cdots \oplus T_d[x_d].$$

Note, in addition, that we might as well assume that $N \leq M^d$, for otherwise we could simply use each key, k, as an integer index for hashing, as in the standard lookup table implementation. Also, recall that the number of keys, n, may be different than the capacity, N, of our hash table.

Because the values in the tables are themselves chosen at random, such a function is itself fairly random. For instance, it can be shown that such a function will cause two distinct keys to collide at the same hash value with probability $1/N$, which is what we would get from a perfectly random function. In addition, this function only uses the XOR operation; hence, it can be evaluated without doing any additions and multiplications, which may take more time than XOR operations on some machines.

6.2.4 Modular Division

In the case that we can view the key k as a single integer, perhaps the simplest hash function is to use modular division to compress k into the range $[0, N-1]$ as follows:

$$h(k) = k \bmod N.$$

If we take N to be a prime number, then the division compression map helps "spread out" the distribution of hashed values. Indeed, if N is not prime, there is a higher likelihood that patterns in the distribution of keys will be repeated in the distribution of hash codes, thereby causing collisions. For example, if we hash the keys $\{200, 205, 210, 215, 220, \ldots, 600\}$ to a bucket array of size 100, then each hash code will collide with three others. But if this same set of keys is hashed to a bucket array of size 101, then there will be no collisions. If we can assume that the set of keys is uniformly randomly distributed among an integer range much larger than N, then the modular-division hash function should guarantee that the probability of two different keys being hashed to the same value is at most $1/N$.

Choosing N to be a prime number and having uniformly distributed keys is not always enough to avoid such collisions, however, if the keys are not also random. For example, if there is a repeated pattern of key values of the form $iN + j$ for several different i's, then such a set of keys would be uniformly distributed but they will all collide at the hash value j.

6.2.5 Random Linear and Polynomial Functions

A more sophisticated compression function, which helps mitigate repeated patterns in a set of integer keys is to use a ***random linear hash function***. In this case, we define a ***random linear hash function***, h, for an integer key, k, as

$$h(k) = (ak + b) \bmod N,$$

where N is a prime number, and $0 < a < N$ and $0 \le b < N$ are independent uniform random integers. This hash function is chosen in order to reduce the number of collisions caused by repeated patterns in a set of hash values and thereby get us closer to having a "good" hash function, that is, one where the probability any two different keys collide is $1/N$. In fact, we prove below, in Section 6.5, that a random linear hash function can be used to guarantee such a probability bound on pairwise collisions.

Random Polynomial Functions

In some cases, we may need a hash function that guarantees that the probabilities that larger numbers of hash values collide is the same as what one would get with a random function. One way to achieve such a result for an integer key, k, is to use a ***random polynomial hash function***, h, which is defined as

$$h(k) = a_d + k(a_{d-1} + k(a_{d-2} + \cdots + k(a_3 + k(a_2 + ka_1))\cdots)) \bmod N,$$

where N is prime and the a_i's are independent uniformly random integers in the range $[0, N-1]$, such that at least one of the coefficients in $\{a_1, a_2, \ldots, a_{d-1}\}$ is nonzero. For such a hash function, although we don't include a proof here, one can show that the probability that any $2 \le c \le d$ distinct keys collide at the same hash value is $1/N^{c-1}$.

Thus, adding this hash function to the bunch we have already discussed, we clearly have a rich collection of hash functions, each of which is designed to minimize the number of collisions. Even so, it is not likely that we can completely avoid collisions with any of these schemes. Therefore, in addition to choosing a good hash function, we also need to come up with a graceful way of handling collisions when they occur.

6.3 Handling Collisions and Rehashing

Recall that the main idea of a hash table is to take a lookup table, A, and a hash function, h, and use them to implement a map by storing each item (k, v) in the "bucket" $A[h(k)]$. This simple idea is challenged, however, when we have two distinct keys, k_1 and k_2, such that $h(k_1) = h(k_2)$. The existence of such ***collisions*** prevents us from simply inserting a new item (k, v) directly in the bucket $A[h(k)]$. They also complicate our procedure for performing the get(k) operation. Thus, we need consistent strategies for resolving collisions.

6.3.1 Separate Chaining

A simple and efficient way for dealing with collisions is to have each bucket $A[i]$ store a reference to a set, S_i, that stores all the items that our hash function has mapped to the bucket $A[i]$ in a linked list. The set S_i can be viewed as a miniature map, implemented using the underlying linked list, but restricted to only hold items (k, v) such that $h(k) = i$. This ***collision resolution*** rule is known as ***separate chaining***. Assuming that we implement each nonempty bucket in this way, we can perform the fundamental map operations as follows:

- get(k):

```
$B \leftarrow A[h(k)]$
if $B =$ NULL then
    return NULL
else
    return $B$.get$(k)$   // do a lookup in the list $B$
```

- put(k, v):

```
if $A[h(k)] =$ NULL then
    Create a new initially empty linked-list-based map, $B$
    $A[h(k)] \leftarrow B$
else
    $B \leftarrow A[h(k)]$
$B$.put$(k, v)$   // put $(k, v)$ at the end of the list $B$
```

- remove(k):

```
$B \leftarrow A[h(k)]$
if $B =$ NULL then
    return NULL
else
    return $B$.remove$(k)$   // remove the item with key $k$ from the list $B$
```

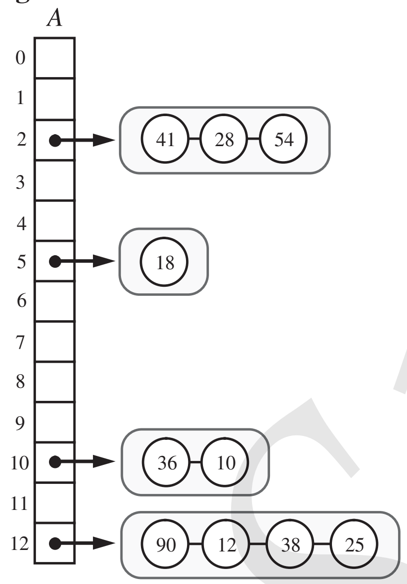


Figure 6.4: Example of a hash table of size 13, storing 10 integer keys, with collisions resolved by the chaining method. The hash function in this case is $h(k) = k \bmod 13$.

Thus, for each map operation involving a key k, we delegate the handling of this operation to the miniature linked-list-based map stored at $A[h(k)]$. So, an insertion will put the new item at the beginning of this list in $O(1)$ time, a find will search through this sequence until it reaches the end or finds an item with the desired key, and a remove will additionally remove an item after it is found. We can "get away" with using the simple linked-list implementation in these cases, because the spreading properties of the hash function help keep each such list small. Indeed, a good hash function will try to minimize collisions as much as possible, which will imply that most of our buckets are either empty or store just a single item.

In Figure 6.4, we give an illustration of a simple hash table that uses the modular division hash function and separate chaining to resolve collisions.

For the sake of analyzing separate chaining, let us assume that our hash function, h, maps keys to independent uniform random values in the range $[0, N-1]$. Thus, if we let X be a random variable representing the number of items that map to a bucket, i, in the array A, then the expected value of X,

$$E(X) = \frac{n}{N},$$

where n is the number of items in the map, since each of the N locations in A is equally likely for each item to be placed. This parameter, n/N, which is the ratio of the number of items in a hash table, n, and the capacity of the table, N, is called the ***load factor*** of the hash table. If it is $O(1)$, then the above analysis says that the expected time for hash table operations is $O(1)$ when collisions are handled with separate chaining.

6.3.2 Open Addressing

The separate chaining rule has many nice properties, such as allowing for simple implementations of the fundamental map operations, but it nevertheless has one disadvantage: it requires the use of auxiliary data structures—namely, the linked lists that store the items for each index in the array. We can handle collisions in other ways besides using the separate chaining rule, however. In particular, if space is of a premium, then we can use the alternative approach of always storing each item directly in a bucket, at most one item per bucket. This approach saves space because no auxiliary structures are employed—it only uses the space in the bucket array, A, itself—but it requires a bit more complexity to deal with collisions. There are several methods for implementing this approach, which is referred to as ***open addressing***.

6.3.3 Linear Probing

One of the simplest open addressing collision-handling strategy is ***linear probing***. In this strategy, if we try to insert an item (k, v) into a bucket $A[i]$ that is already occupied, where $i = h(k)$, then we try next at $A[(i+1) \bmod N]$. If $A[(i+1) \bmod N]$ is occupied, then we try $A[(i+2) \bmod N]$, and so on, until we find an empty bucket in A that can accept the new item. Once this bucket is located, we simply insert the item (k, v) here. Of course, using this collision resolution strategy requires that we change the implementation of the get(k) operation. In particular, to perform such a search we must examine consecutive buckets, starting from $A[h(k)]$, until we either find an item with key equal to k or we find an empty bucket (in which case the search is unsuccessful). (See Figure 6.5.)

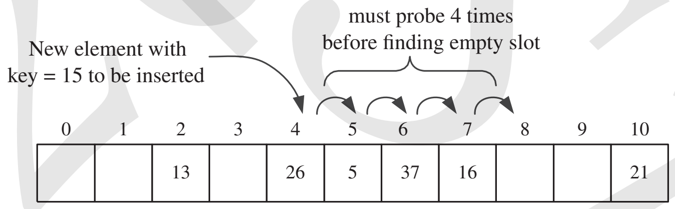


Figure 6.5: An insertion into a hash table using linear probing to resolve collisions. Here we use the compression map $h(k) = k \bmod 11$.

The operation remove(k) is more complicated, however. In particular, to fully implement this method, we should restore the contents of the bucket array to look as though the item with key k was never inserted in its bucket $A[i]$ in the first place. The algorithms for the three fundamental map methods are as follows, where, for the sake of simplicity, we assume that the table, A, is not completely full.

- get(k):

```
$i \leftarrow h(k)$
**while** $A[i] \neq$ NULL **do**
    **if** $A[i]$.key $= k$ **then**
        **return** $A[i]$
    $i \leftarrow (i+1) \bmod N$
**return** NULL
```

- put(k, v):

```
$i \leftarrow h(k)$
**while** $A[i] \neq$ NULL **do**
    **if** $A[i]$.key $= k$ **then**
        $A[i] \leftarrow (k, v)$        // replace the old $(k, v')$
    $i \leftarrow (i+1) \bmod N$
$A[i] \leftarrow (k, v)$
```

- remove(k):

```
$i \leftarrow h(k)$
**while** $A[i] \neq$ NULL **do**
    **if** $A[i]$.key $= k$ **then**
        $temp \leftarrow A[i]$
        $A[i] \leftarrow$ NULL
        Call Shift($i$) to restore $A$ to a stable state without $k$
        **return** $temp$
    $i \leftarrow (i+1) \bmod N$
**return** NULL
```

- Shift(i):

```
$s \leftarrow 1$        // the current shift amount
**while** $A[(i+s) \bmod N] \neq$ NULL **do**
    $j \leftarrow h(A[(i+s) \bmod N]$.key$)$        // preferred index for this item
    **if** $j \notin (i, i+s] \bmod N$ **then**
        $A[i] \leftarrow A[(i+s) \bmod N]$        // fill in the "hole"
        $A[(i+s) \bmod N] \leftarrow$ NULL        // move the "hole"
        $i \leftarrow (i+s) \bmod N$
        $s \leftarrow 1$
    **else**
        $s \leftarrow s+1$
```

One alternative to the shifting done above for remove(k) is to replace the deleted item by a special "deactivated item" object. With this special marker possibly occupying buckets in our hash table, we would then need to modify our search algorithm for remove(k) or get(k), so that the search for a key k should skip over deactivated items and continue probing until reaching the desired item or an empty bucket. Likewise, the put(k, v) algorithm should stop at a deactivated item and replace it with the new item to be inserted. (See Exercise C-6.1.)

Analyzing Linear Probing

Recall that, in the linear-probing scheme for handling collisions, whenever an insertion at a cell i would cause a collision, then we instead insert the new item in the first cell of $i+1$, $i+2$, and so on, until we find an empty cell, wrapping our indices to the beginning of the table if necessary. In order to analyze how long this takes on average, we are going to use one of the Chernoff bounds, which we also discuss in Sections 1.2.4 and 19.5.

Let $X_1, X_2, \ldots, X_n$ be a set of mutually independent indicator random variables, such that each X_i is 1 with some probability $p_i > 0$ and 0 otherwise. Let $X = \sum_{i=1}^{n} X_i$ be the sum of these random variables, and let μ denote the mean of X, that is, $\mu = E(X) = \sum_{i=1}^{n} p_i$. The following bound, which is due to Chernoff (and which we derive in Section 19.5), establishes that, for $\delta > 0$,

$$\Pr(X > (1+\delta)\mu) < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu}.$$

Having presented this Chernoff bound, we can now analyze the expected running time for doing a search or update in a hash table that is implemented using the linear-probing collision-handling scheme. For this analysis, let us assume that we are storing n items in a hash table of size $N = 2n$, that is, our hash table has a load factor of $1/2$. If our actual load factor is less than this, then this analysis still applies, of course. If the load factor is much more than this, however, then the performance of linear probing can degrade significantly.

Let X denote a random variable equal to the number of probes that we would perform in doing a search or update operation in our hash table for some key, k. Furthermore, let X_i be a 0/1 indicator random variable that is 1 if and only if $i = h(k)$, and let Y_i be a random variable that is equal to the length of a run of contiguous nonempty cells that begins at position i, wrapping around the end of the table if necessary. By the way that linear probing works, and because we assume that our hash function $h(k)$ is random,

$$X = \sum_{i=0}^{N-1} X_i(Y_i + 1),$$

which implies that

$$\begin{aligned} E(X) &= \sum_{i=0}^{N-1} \frac{1}{2n} E(Y_i + 1) \\ &= 1 + (1/2n) E\left(\sum_{i=0}^{N-1} Y_i\right). \end{aligned}$$

Thus, if we can bound the expected value of the sum of Y_i's, then we can bound the expected time for a search or update operation in a linear-probing hashing scheme.

Consider, then, a maximal contiguous sequence, S, of k nonempty table cells, that is, a contiguous group of occupied cells that has empty cells next to its opposite ends. Any search or update operation that lands in S will, in the worst case, march all the way to the end of S. That is, if a search lands in the first cell of S, it would make k wasted probes, if it lands in the second cell of S, it would make $k-1$ wasted probes, and so on. So the total cost of all the searches that land in S can be at most k^2. Thus, if we let $Z_{i,k}$ be a 0/1 indicator random variable for the existence of a maximal sequence of nonempty cells of length k, then

$$\sum_{i=0}^{N-1} Y_i \le \sum_{i=0}^{N-1} \sum_{k=1}^{2n} k^2 Z_{i,k}.$$

Put another way, it is as if we are "charging" each maximal sequence of nonempty cells for all the searches that land in that sequence.

So, to bound the expected value of the sum of the Y_i's, we need to bound the probability that $Z_{i,k}$ is 1, which is something we can do using the Chernoff bound given above. Let Z_k denote the number of items that are mapped to a given sequence of k cells in our table. Then,

$$\Pr(Z_{i,k} = 1) \le \Pr(Z_k \ge k).$$

Because the load factor of our table is $1/2$, $E(Z_k) = k/2$. Thus, by the above Chernoff bound,

$$\begin{aligned} \Pr(Z_k \ge k) &= \Pr(Z_k \ge 2(k/2)) \\ &\le (e/4)^{k/2} \\ &< 2^{-k/4}. \end{aligned}$$

Therefore, putting all the above pieces together,

$$\begin{aligned} E(X) &= 1 + (1/2n) E\left(\sum_{i=0}^{N-1} Y_i\right) \\ &\le 1 + (1/2n) \sum_{i=0}^{N-1} \sum_{k=1}^{2n} k^2\, 2^{-k/4} \\ &\le 1 + \sum_{k=1}^{\infty} k^2\, 2^{-k/4} \\ &= O(1). \end{aligned}$$

That is, the expected running time for doing a search or update operation with linear probing is $O(1)$, so long as the load factor in our hash table is at most $1/2$.

Linear probing saves space, but it admittedly complicates removals. In addition, if the load factor of the hash table goes too high, then the linear-probing collision-handling strategy tends to cluster the items of the map into contiguous runs, which causes searches to slow down.

6.3.4 Quadratic Probing

Another open addressing strategy, known as ***quadratic probing***, involves iteratively trying the buckets $A[(i+f(j)) \bmod N]$, for $j = 0, 1, 2, \ldots$, where $f(j) = j^2$, until finding an empty bucket. As with linear probing, the quadratic probing strategy complicates the removal operation, but it does avoid the kinds of clustering patterns that occur with linear probing.

Secondary Clustering

Nevertheless, when the load factor approaches 1, it creates its own kind of clustering, called ***secondary clustering***, where the set of filled array cells "bounces" around the array in a fixed pattern. If N is not chosen as a prime, then the quadratic probing strategy may not find an empty bucket in A even if one exists. In fact, even if N is prime, this strategy may not find an empty slot, if the bucket array is at least half full.

6.3.5 Double Hashing

Another open addressing strategy that does not cause clustering of the kind produced by linear probing or the kind produced by quadratic probing is the ***double hashing*** strategy. In this approach, we choose a secondary hash function, h', and if h maps some key k to a bucket $A[i]$, with $i = h(k)$, that is already occupied, then we iteratively try the buckets $A[(i + f(j)) \bmod N]$ next, for $j = 1, 2, 3, \ldots$, where $f(j) = j \cdot h'(k)$. In this scheme, the secondary hash function is not allowed to evaluate to zero; a common choice is $h'(k) = q - (k \bmod q)$, for some prime number $q < N$. Also, N should be a prime.

Moreover, in using the double hashing technique, we should choose a secondary hash function that will attempt to minimize clustering as much as possible. If the functions h and f are assumed to be random functions, then it is fairly straightforward to prove that the expected running time for a search is $O(1)$, for example, see Exercise C-6.3.

Trade-offs for Open Addressing

These ***open addressing*** schemes save some space over the separate chaining method, but they are not necessarily faster. In experimental and theoretical analyses, the chaining method is either competitive or faster than the other methods, if the load factor is relatively close to 1, and open addressing tends to be faster than separate chaining if the load factor is less than $1/2$ (because it avoids an extra pointer hop).

6.3.6 Rehashing

As shown above, the ***load factor***, n/N, which is the ratio of the number of items in a hash table, n, and the capacity of the table, N, has a big impact on the performance of a hash table. The load factor of a hash table correlates to the probability that a newly inserted item will collide with an existing item, and, as discussed above, the load factor impacts the running times for hash table operations for both separate chaining and open addressing methods for handling collisions. For instance, given a constant load factor, the above analysis for separate chaining implies that the expected running time of the operations get, put, and remove in a map implemented with a hash table with separate chaining is $O(\lceil n/N \rceil) = O(1)$. Thus, we can implement the standard map operations to run in constant expected time, assuming we can maintain our hash table to have a bounded load factor. Therefore, we should always keep the load factor of our hash table below a small constant sufficiently smaller than 1.

Keeping a hash table's load factor below a constant (like $1/2$ or $3/4$) requires additional work whenever we add an item that would cause us to exceed this bound. In such cases, in order to keep the load factor below the specified constant, we need to increase the size of our bucket array, A, and change our hash function to match this new size. Moreover, we must then insert all the existing hash-table elements into the new bucket array using the new hash function. Such a size increase and hash table rebuild is called ***rehashing***. Following the approach of the extendable array (Section 1.4.2), a good choice is to rehash into an array roughly double the size of the original array. Such a choice implies that the additional amortized cost required for rehashing is $O(1)$ per insertion operation. (See Figure 6.6.)

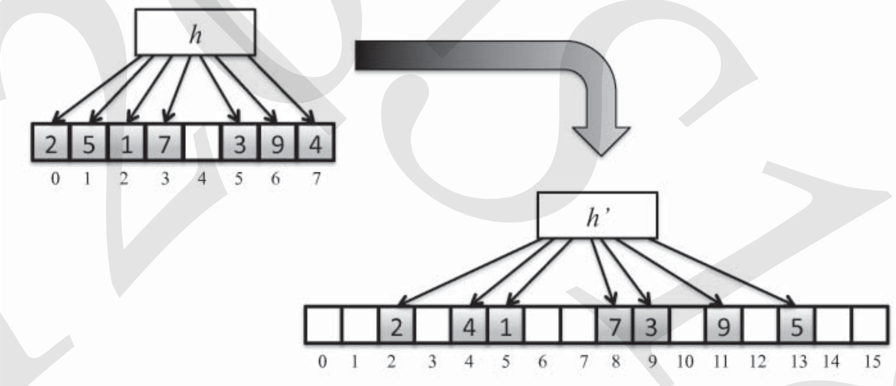


Figure 6.6: Rehashing a hash table with capacity 8 and load factor $7/8$ into a hash table with capacity 16 and load factor $7/16$. Note that all the items are placed into their new locations according to a new hash function, h'. So, for example, the item with key 7 was placed in bucket 3 in the old table, since $h(7) = 3$, but is placed in bucket 8 in the new table, since $h'(7) = 8$. (We only show keys, not values.)

6.4 Cuckoo Hashing

In the separate chaining and open addressing hashing schemes, the running time for doing a search is expected to be $O(1)$, but it can be as bad as $O(n)$ in the worst case (albeit with very low probability). In the case of separate chaining, the running time of the put(k, v) method runs in $O(1)$ time in the worst case, however. In most applications, we would expect to perform more searches than insertions, so it would be nice to have a collision-handling scheme that can guarantee that searches run in $O(1)$ time in the worst case, while allowing for insertions to run in $O(1)$ time as an expected bound. Interestingly, the ***cuckoo hashing*** scheme we describe in this section achieves this performance goal while still being an open addressing scheme, like linear probing.

The Power of Two Choices

In the cuckoo hashing scheme, we use two lookup tables, T_0 and T_1, each of size N, where N is greater than n, the number of items in the map, by at least a constant factor, say, $N \geq 2n$. In addition, we use a hash function, h_0, for T_0, and a different hash function, h_1, for T_1. For any key, k, there are only two possible places where we are allowed to store an item with key k, namely, either in $T_0[h_0(k)]$ or $T_1[h_1(k)]$. (See Figure 6.7.)

The way we perform the get(k) method in this scheme is quite simple:

```
• get($k$):
      if $T_0[h_0(k)] \neq$ NULL  and  $T_0[h_0(k)]$.key $= k$ then
          return $T_0[h_0(k)]$
      if $T_1[h_1(k)] \neq$ NULL  and  $T_1[h_1(k)]$.key $= k$ then
          return $T_1[h_1(k)]$
      return NULL
```

This is clearly a constant-time operation, and performing the remove(k) operation is similar:

```
• remove($k$):
      if $T_0[h_0(k)] \neq$ NULL  and  $T_0[h_0(k)]$.key $= k$ then
          $temp \leftarrow T_0[h_0(k)]$
          $T_0[h_0(k)] \leftarrow$ NULL
          return $temp$
      if $T_1[h_1(k)] \neq$ NULL  and  $T_1[h_1(k)]$.key $= k$ then
          $temp \leftarrow T_1[h_1(k)]$
          $T_1[h_1(k)] \leftarrow$ NULL
          return $temp$
      return NULL
```

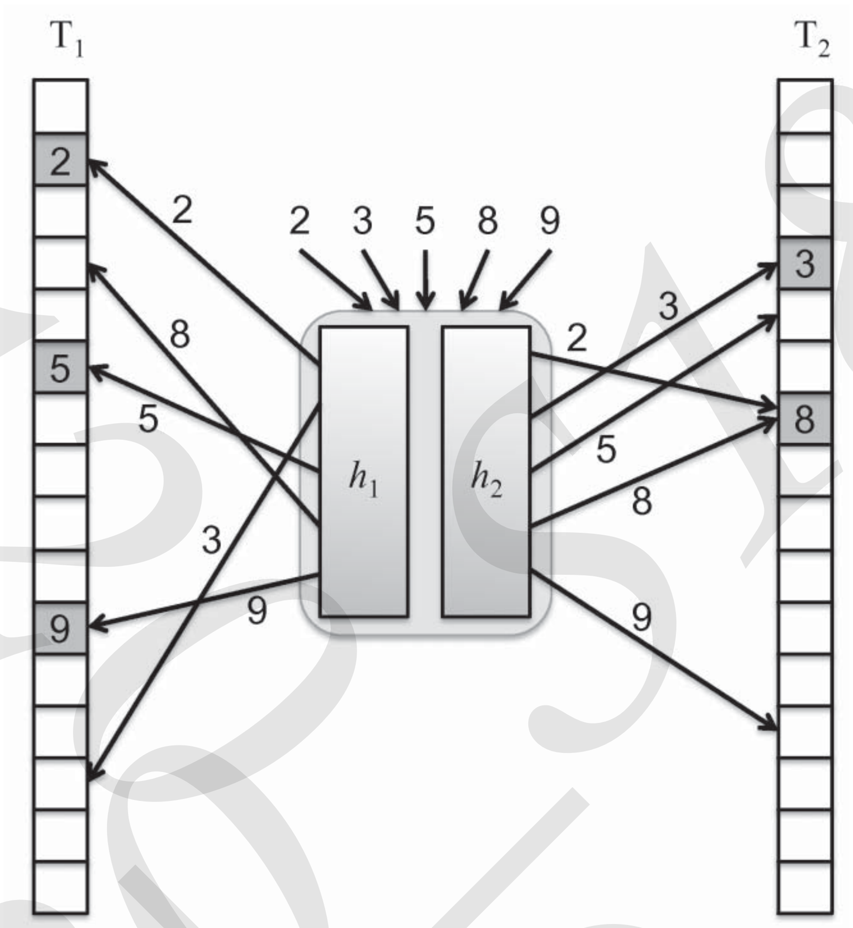


Figure 6.7: Cuckoo hashing. Each of the keys in the set $S = \{2, 3, 5, 8, 9\}$ has two possible locations it can go, one in the table T_1 and one in the table T_2. Note that there is a collision of 2 and 8 in T_2, but that is okay, since there is no collision for 2 in its alternative location, in T_1.

The name "cuckoo hashing" comes from the way the put(k, v) operation is performed in this scheme, because it mimics the breeding habits of the Common Cuckoo bird. The Common Cuckoo is a brood parasite—it lays its egg in the nest of another bird after first evicting an egg out of that nest. Similarly, if a collision occurs in the insertion operation in the cuckoo hashing scheme, then we evict the previous item in that cell and insert the new one in its place. This forces the evicted item to go to its alternate location in the other table and be inserted there, which may repeat the eviction process with another item, and so on. Eventually, we either find an empty cell and stop or we repeat a previous eviction, which indicates an eviction cycle. If we discover an eviction cycle, then we stop the insertion process and rehash all the items in the two tables using new, hopefully better, hash functions.

Insertions

The pseudocode for the put(k, v) method is as follows. (See Figure 6.8.)

```
• put(k, v):
      **if** $T_0[h_0(k)] \neq$ NULL **and** $T_0[h_0(k)]$.key $= k$ **then**
          $T_0[h_0(k)] \leftarrow (k, v)$
          **return**
      **if** $T_1[h_1(k)] \neq$ NULL **and** $T_1[h_1(k)]$.key $= k$ **then**
          $T_1[h_1(k)] \leftarrow (k, v)$
          **return**
      $i \leftarrow 0$
      **repeat**
          **if** $T_i[h_i(k)] =$ NULL **then**
              $T_i[h_i(k)] \leftarrow (k, v)$
              **return**
          $temp \leftarrow T_i[h_i(k)]$
          $T_i[h_i(k)] \leftarrow (k, v)$        // cuckoo eviction
          $(k, v) \leftarrow temp$
          $i \leftarrow (i + 1) \bmod 2$
      **until** a cycle occurs
      Rehash all the items, plus $(k, v)$, using new hash functions, $h_0$ and $h_1$.
```

Note that the above pseudocode has a condition for detecting a cycle in the sequence of insertions. There are several ways to formulate this condition. For example, we could count the number of iterations for this loop and consider there to be a cycle if we go over a certain threshold, like n or $\log n$.

Analysis of Cuckoo Hashing

Let us analyze the expected running time for doing an insertion in the cuckoo hashing scheme. Throughout this analysis, we assume $N \geq 2n$, where N is the size of each table and n is the number of items in our map. Central to this analysis is an analysis of the possibility that a sequence of evictions could start at a cell, x_1, and evict an item that goes to a cell, x_2, and evicts an item that goes to a cell, x_3, and so on. Ignoring the direction that such a sequence of evictions goes in, say that there is a potential sequence of evictions of ***length*** 1 between x and y if there is an item that maps to both x and y as its two possible locations. Likewise, say that there is a potential sequence of evictions of ***length*** L between x and y if there is a possible sequence of evictions of length $L - 1$ between x and some cell, z, and there is also an item that maps to both z and y as its two possible locations. We begin our analysis with a useful fact about the probability that there will be a possible sequence of evictions of length L between a cell, x, and a cell, y, in $T_0 \cup T_1$.

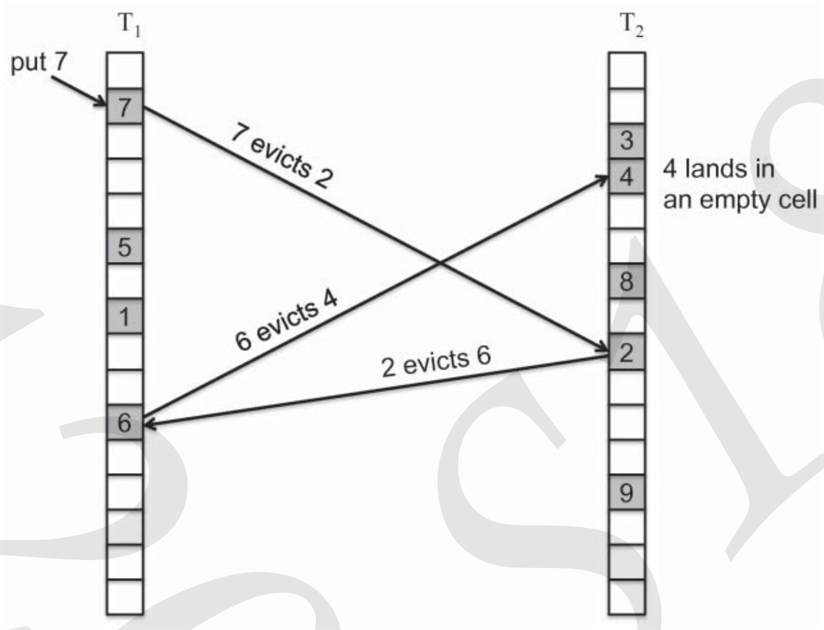


Figure 6.8: An eviction sequence of length 3.

Bounding the Probability of Long Eviction Sequences

The following lemma gives us a way of bounding the probability that we might have long eviction sequences for any put operation.

Lemma 6.1: *The probability that there is a possible sequence of evictions of length L between a cell, x, and a cell, y, in $T_0 \cup T_1$, is at most $1/(2^L N)$.*

Proof: The proof is by induction. For the base case, $L = 1$, note that there is a possible length-1 eviction sequence between x and y if and only if we choose x and y as the two possible locations for some item. The hash functions h_0 and h_1 effectively choose 1 cell each out of N possible cells in T_0 and T_1, respectively. Thus, the probability that a particular item chooses both x and y as its two locations is at most $1/N^2$, and the probability that ***any*** of the n items in our set, S, chooses both x and y is at most

$$\begin{aligned} \sum_{(k,v)\in S} \frac{1}{N^2} &= \frac{n}{N^2} \\ &\leq \frac{1}{2}\cdot\frac{1}{N}, \end{aligned}$$

because of our assumption that $N \geq 2n$. Note that the probability that there is a possible eviction sequence of length 1 between x and y is 0 if both x and y are in the same table, T_i, but this probability is clearly bounded by $1/(2N)$. So this completes the proof for the base case, $L = 1$.

For the inductive step, $L \geq 2$, let us assume the claim is true for possible eviction sequences of length $(L-1)$. For there to be a length-L eviction sequence from x to y, there has to be a possible length-$(L-1)$ eviction sequence between x and some cell, z, and a possible length-1 eviction sequence between z and y. By induction, for a given cell, z, the probability there is a length-$(L-1)$ eviction sequence between x and z is at most $1/(2^{L-1}N)$. Likewise, from the above discussion for the base case, there is a length-1 eviction sequence between z and y with probability at most $1/(2N)$. Thus, the probability that there is a length-L eviction sequence between x and y is at most

$$\begin{aligned}\sum_z \frac{1}{2^{L-1}N} \cdot \frac{1}{2N} &= \sum_z \frac{1}{2^L N^2} \\ &\leq \frac{N}{2^L N^2} \\ &= \frac{1}{2^L N},\end{aligned}$$

since there are only N candidates for z, because it has to be in either T_0 or T_1, depending, respectively, on whether y is in T_1 or T_0. ■

Counting the Expected Number of Possible Evictions

Say that two keys k and l are in the same "bucket" if there is a sequence of evictions (of any length) between a possible cell for k and a possible cell for l. Thus, for k and l to be in the same bucket, there has to be an eviction sequence between one of the cells $T_0[h_0(k)]$ or $T_1[h_1(k)]$ and one of the cells $T_0[h_0(l)]$ or $T_1[h_1(l)]$. Note that there are 4 such possible sequences, depending on where we stop and end, since we are ignoring the direction that a sequence of evictions can take. Then, by Lemma 6.1, and summing across all possible lengths, the probability that k and l are in the same bucket is bounded by

$$\begin{aligned}4\sum_{L=1}^{\infty} \frac{1}{2^L N} &= \frac{4}{N}\sum_{L=1}^{\infty} \frac{1}{2^L} \\ &= \frac{4}{N}.\end{aligned}$$

Note that, by this notion of a "bucket," the running time for performing an insertion in a cuckoo table is certainly bounded by the number of items that map to the same bucket as the item being inserted, so long as we don't cause a rehash. Thus, the expected time to insert an item with key k in this case is bounded by

$$\sum_{\text{keys in } S} \frac{4}{N} = 4n/N \leq 2.$$

In other words, the expected running time for performing a put(k, v) operation for a cuckoo table is $O(1)$, assuming this insertion does not cause a rehash.

Allowing for Rehash Operations

So, let us consider the expected number of rehashes we may have to do. By the description of the insertion algorithm, a rehash occurs if there is a cycle of evictions, that is, a length-L sequence of evictions that starts and ends at the same cell. Then, by Lemma 6.1, and the fact that we need only consider even-length sequences of evictions (to form a cycle), the probability that there is a cycle of evictions that starts and ends at some cell, x, is bounded by

$$\begin{aligned}\sum_{L=1}^{\infty} \frac{1}{2^{2L}N} &= \sum_{L=1}^{\infty} \frac{1}{4^{L}N} \\ &= \frac{1}{N}\sum_{L=1}^{\infty} \frac{1}{4^{L}} \\ &= \frac{1}{N} \cdot \frac{1/4}{1-1/4} \\ &= \frac{1}{3N}.\end{aligned}$$

Therefore, the probability that there is a cycle anywhere can be bounded by summing this value over all $2N$ cells in the cuckoo tables. That is, the probability that any of the n items in our cuckoo hash table forms a cycle of evictions is at most $2/3$. In other words, with probability at most $2/3$ we will have to do a rehash during the insertion of these n items, and, with probability at most $(2/3)^2$ we would have to do 2 rehashes, and, with probability at most $(2/3)^3$ we would have to do 3 rehashes, and so on. If the time to do a rehash is $O(n)$, then the expected time to perform n insertions and the rehashes they may cause is bounded by

$$O(n) + O(n) \cdot \sum_{i=1}^{\infty} (2/3)^i,$$

which is $O(n)$. Thus, the expected amortized time to perform any single insertion in a cuckoo table is $O(1)$.

Summary

So, to sum up, a cuckoo hash table achieves worst-case constant time for lookups and removals, and expected constant time for insertions. This is primarily because there are exactly two possible places for any item to be in a cuckoo hash table, which shows the power of two choices.

6.5 Universal Hashing

In this section, we show how a random linear hash function can be probabilistically shown to be good. Recall that in this case we assume that our set of keys are integers in some range. Let $[0, M-1]$ be this range. Thus, we can view a hash function h as a mapping from integers in the range $[0, M-1]$ to integers in the range $[0, N-1]$, and we can view the set of candidate hash functions we are considering as a ***family*** H of hash functions. Such a family is ***universal*** if for any two integers j and k in the range $[0, M-1]$ and for a hash function chosen uniformly at random from H,

$$\Pr(h(j) = h(k)) \leq \frac{1}{N}.$$

Such a family is also known as a ***2-universal*** family of hash functions. The goal of choosing a good hash function can therefore be viewed as the problem of selecting a small universal family H of hash functions that are easy to compute. The reason universal families of hash functions are useful is that they result in a low expected number of collisions.

Theorem 6.2: *Let j be an integer in the range $[0, M-1]$, let S be a set of n integers in this same range, and let h be a hash function chosen uniformly, at random, from a universal family of hash functions from integers in the range $[0, M-1]$ to integers in the range $[0, N-1]$. Then the expected number of collisions between j and the integers in S is at most n/N.*

Proof: Let $c_h(j, S)$ denote the number of collisions between j and integers in S (that is, $c_h(j, S) = |\{k \in S\colon h(j) = h(k)\}|$). The quantity we are interested in is the expected value $E(c_h(j, S))$. We can write $c_h(j, S)$ as

$$c_h(j, S) = \sum_{k \in S} X_{j,k},$$

where $X_{j,k}$ is a random variable that is 1 if $h(j) = h(k)$ and is 0 otherwise (that is, $X_{j,k}$ is an ***indicator*** random variable for a collision between j and k). By the linearity of expectation,

$$E(c_h(j, S)) = \sum_{s \in S} E(X_{j,k}).$$

Also, by the definition of a universal family, $E(X_{j,k}) \leq 1/N$. Thus,

$$E(c_h(j, S)) \leq \sum_{s \in S} \frac{1}{N} = \frac{n}{N}.$$

■

Put another way, this theorem states that the expected number of collisions between a hash code j and the keys already in a hash table (using a hash function chosen at random from a universal family H) is at most the current load factor of the hash table. Since the time to perform a search, insertion, or deletion for a key j in a hash table that uses the chaining collision-resolution rule is proportional to the number of collisions between j and the other keys in the table, this implies that the expected running time of any such operation is proportional to the hash table's load factor. This is exactly what we want.

Let us turn our attention, then, to the problem of constructing a small universal family of hash functions that are easy to compute. The set of hash functions we construct is actually similar to the final family we considered at the end of the previous section. Let p be a prime number greater than or equal to the number of hash codes M but less than $2M$ (and there must always be such a prime number, according to a mathematical fact known as ***Bertrand's Postulate***).

Define H as the set of hash functions of the form

$$h_{a,b}(k) = (ak + b \bmod p) \bmod N.$$

The following theorem establishes that this family of hash functions is universal.

Theorem 6.3: *The family $H = \{h_{a,b} \colon 0 < a < p$ and $0 \le b < p\}$ is universal.*

Proof: Let Z denote the set of integers in the range $[0, p-1]$. Let us separate each hash function $h_{a,b}$ into the functions

$$f_{a,b}(k) = (ak + b) \bmod p$$

and

$$g(k) = k \bmod N,$$

so that $h_{a,b}(k) = g(f_{a,b}(k))$. The set of functions $f_{a,b}$ defines a family of hash functions F that map integers in Z to integers in Z. We claim that each function in F causes no collisions at all. To justify this claim, consider $f_{a,b}(j)$ and $f_{a,b}(k)$ for some pair of different integers j and k in Z. If $f_{a,b}(j) = f_{a,b}(k)$, then we would have a collision. But, recalling the definition of the modulo operation, this would imply that

$$aj + b - \left\lfloor \frac{aj+b}{p} \right\rfloor p = ak + b - \left\lfloor \frac{ak+b}{p} \right\rfloor p.$$

Without loss of generality, we can assume that $k < j$, which implies that

$$a(j-k) = \left(\left\lfloor \frac{aj+b}{p} \right\rfloor - \left\lfloor \frac{ak+b}{p} \right\rfloor \right) p.$$

Since $a \neq 0$ and $k < j$, this in turn implies that $a(j-k)$ is a multiple of p. But $a < p$ and $j - k < p$, so there is no way that $a(j-k)$ can be a positive multiple of p, because p is prime (remember that every positive integer can be factored into a product of primes). So it is impossible for $f_{a,b}(j) = f_{a,b}(k)$ if $j \neq k$. To put this another way, each $f_{a,b}$ maps the integers in Z to the integers in Z in a way that defines a one-to-one correspondence. Since the functions in F cause no collisions, the only way that a function $h_{a,b}$ can cause a collision is for the function g to cause a collision.

Let j and k be two different integers in Z. Also, let $c(j,k)$ denote the number of functions in H that map j and k to the same integer (that is, that cause j and k to collide). We can derive an upper bound for $c(j,k)$ by using a simple counting argument. If we consider any integer x in Z, there are p different functions $f_{a,b}$ such that $f_{a,b}(j) = x$ (since we can choose a b for each choice of a to make this so). Let us now fix x and note that each such function $f_{a,b}$ maps k to a unique integer

$$y = f_{a,b}(k)$$

in Z with $x \neq y$. Moreover, of the p different integers of the form $y = f_{a,b}(k)$, there are at most

$$\lceil p/N \rceil - 1$$

such that $g(y) = g(x)$ and $x \neq y$ (by the definition of g). Thus, for any x in Z, there are at most $\lceil p/N \rceil - 1$ functions $h_{a,b}$ in H such that

$$x = f_{a,b}(j) \quad \text{and} \quad h_{a,b}(j) = h_{a,b}(k).$$

Since there are p choices for the integer x in Z, the above counting arguments imply that

$$\begin{aligned} c(j,k) &\leq p\left(\left\lceil \frac{p}{N} \right\rceil - 1\right) \\ &\leq \frac{p(p-1)}{N}. \end{aligned}$$

There are $p(p-1)$ functions in H, since each function $h_{a,b}$ is determined by a pair (a,b) such that $0 < a < p$ and $0 \leq b < p$. Thus, picking a function uniformly at random from H involves picking one of $p(p-1)$ functions. Therefore, for any two different integers j and k in Z,

$$\begin{aligned} \Pr(h_{a,b}(j) = h_{a,b}(k)) &\leq \frac{p(p-1)/N}{p(p-1)} \\ &= \frac{1}{N}. \end{aligned}$$

That is, the family H is universal. ■

In addition to being universal, the functions in H have a number of other nice properties. Each function in H is easy to select, since doing so simply requires that we select a pair of random integers a and b such that $0 < a < p$ and $0 \leq b < p$. In addition, each function in H is easy to compute in $O(1)$ time, requiring just one multiplication, one addition, and two applications of the modulus function. Thus, any hash function chosen uniformly at random in H will result in an implementation of a map so that the fundamental operations all have expected running times that are $O(\lceil n/N \rceil)$, since we are using the chaining rule for collision resolution.