11 Hash Tables

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list—$\Theta(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$. Indeed, the built-in dictionaries of Python are implemented with hash tables.

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array takes advantage of the $O(1)$ access time for any array element. Section 11.1 discusses direct addressing in more detail. To use direct addressing, you must be able to allocate an array that contains a position for every possible key.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, we *compute* the array index from the key. Section 11.2 presents the main ideas, focusing on "chaining" as a way to handle "collisions," in which more than one key maps to the same array index. Section 11.3 describes how to compute array indices from keys using hash functions. We present and analyze several variations on the basic theme. Section 11.4 looks at "open addressing," which is another way to deal with collisions. The bottom line is that hashing is an extremely effective and practical technique: the basic dictionary operations require only $O(1)$ time on the average. Section 11.5 discusses the hierarchical memory systems of modern computer systems have and illustrates how to design hash tables that work well in such systems.

11.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a distinct key drawn from the universe $U = \{0, 1, \ldots, m - 1\}$, where m is not too large.

To represent the dynamic set, you can use an array, or ***direct-address table***, denoted by $T[0 : m - 1]$, in which each position, or ***slot***, corresponds to a key in the universe U. Figure 11.1 illustrates this approach. Slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] =$ NIL.

The dictionary operations DIRECT-ADDRESS-SEARCH, DIRECT-ADDRESS-INSERT, and DIRECT-ADDRESS-DELETE on the following page are trivial to implement. Each takes only $O(1)$ time.

For some applications, the direct-address table itself can hold the elements in the dynamic set. That is, rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object, save space by storing the object directly in the slot. To indicate an empty slot, use a special key. Then again, why store the key of the object at all? The index of the object *is* its key! Of course, then you'd need some way to tell whether slots are empty.

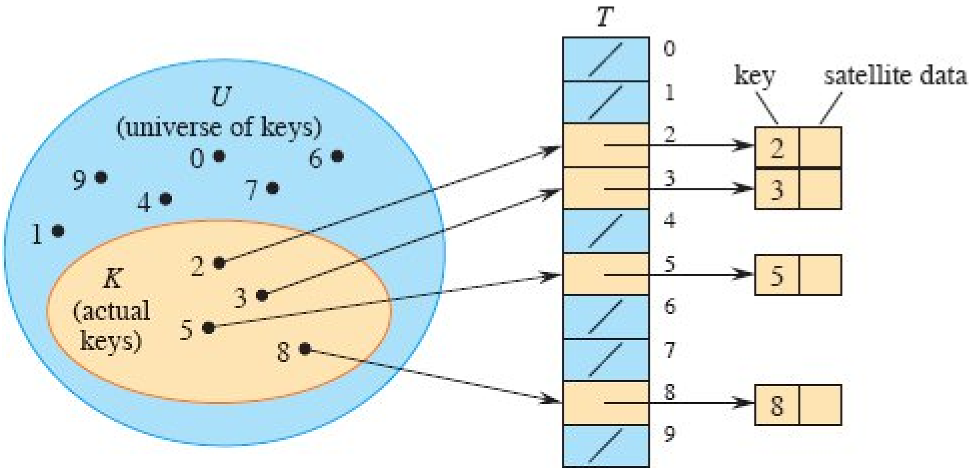

Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index into the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, in blue, contain NIL.

DIRECT-ADDRESS-SEARCH(T, k)
return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
$T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)
$T[x.key] =$ NIL

Exercises

11.1-1

A dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

11.1-2

A ***bit vector*** is simply an array of bits (each either 0 or 1). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements drawn from the set $\{0, 1, \ldots, m-1\}$ and with no satellite data. Dictionary operations should run in $O(1)$ time.

11.1-3

Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

★ ***11.1-4***

Suppose that you want to implement a dictionary by using direct addressing on a *huge* array. That is, if the array size is m and the dictionary contains at most n elements at any one time, then $m \gg n$. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. (*Hint:* Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

11.2 Hash tables

The downside of direct addressing is apparent: if the universe U is large or infinite, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, the storage requirement reduces to $\Theta(|K|)$ while maintaining the benefit that searching for an element in the hash table still requires only $O(1)$ time. The catch is that this bound is for the *average-case time*,[1] whereas for direct addressing it holds for the *worst-case time*.

With direct addressing, an element with key k is stored in slot k, but with hashing, we use a ***hash function*** h to compute the slot number from the key k, so that the element goes into slot $h(k)$. The hash function h maps the universe U of keys into the slots of a ***hash table*** $T[0 : m - 1]$:

$$h : U \rightarrow \{0, 1, \ldots, m - 1\},$$

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k ***hashes*** to slot $h(k)$, and we also say that $h(k)$ is the ***hash value*** of key k. Figure 11.2 illustrates the basic idea. The hash function reduces the range of array indices and hence the size of the array. Instead of a size of $|U|$, the array can have size m. An example of a simple, but not particularly good, hash function is $h(k) = k \bmod m$.

There is one hitch, namely that two keys may hash to the same slot. We call this situation a ***collision***. Fortunately, there are effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution is to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be "random," thus avoiding collisions or at least minimizing their number. The very term "to hash," evoking images of random mixing and chopping, captures the spirit of this approach. (Of course, a hash function h must be deterministic in that a given input k must always produce the same output $h(k)$.) Because $|U| > m$, however, there must be at least two keys that have the same hash value, and avoiding collisions altogether is impossible. Thus, although a well-designed, "random"-looking hash function can reduce the number of collisions, we still need a method for resolving the collisions that do occur.

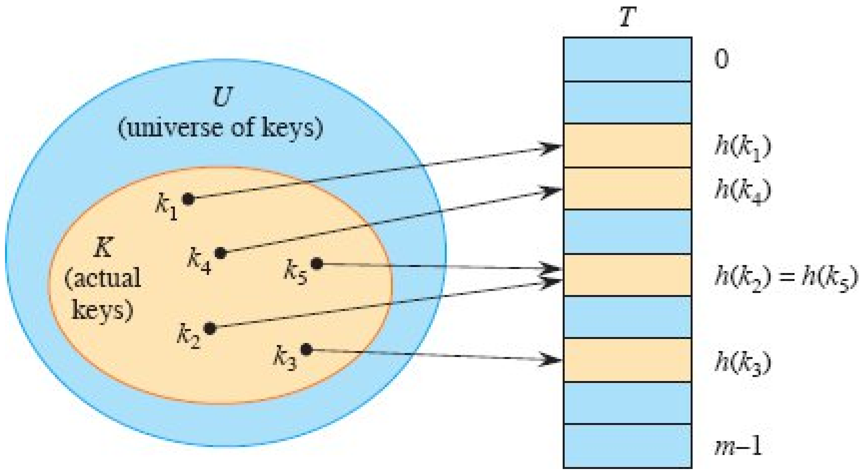

Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

The remainder of this section first presents a definition of "independent uniform hashing," which captures the simplest notion of what it means for a hash function to be "random." It then presents and analyzes the simplest collision resolution technique, called chaining. Section 11.4 introduces an alternative method for resolving collisions, called open addressing.

Independent uniform hashing

An "ideal" hashing function h would have, for each possible input k in the domain U, an output $h(k)$ that is an element randomly and independently chosen uniformly from the range $\{0, 1, \ldots, m - 1\}$. Once a value $h(k)$ is randomly chosen, each subsequent call to h with the same input k yields the same output $h(k)$.

We call such an ideal hash function an ***independent uniform hash function***. Such a function is also often called a ***random oracle*** [43]. When hash tables are implemented with an independent uniform hash function, we say we are using ***independent uniform hashing***.

Independent uniform hashing is an ideal theoretical abstraction, but it is not something that can reasonably be implemented in practice. Nonetheless, we'll analyze the efficiency of hashing under the assumption of independent uniform hashing and then present ways of achieving useful practical approximations to this ideal.

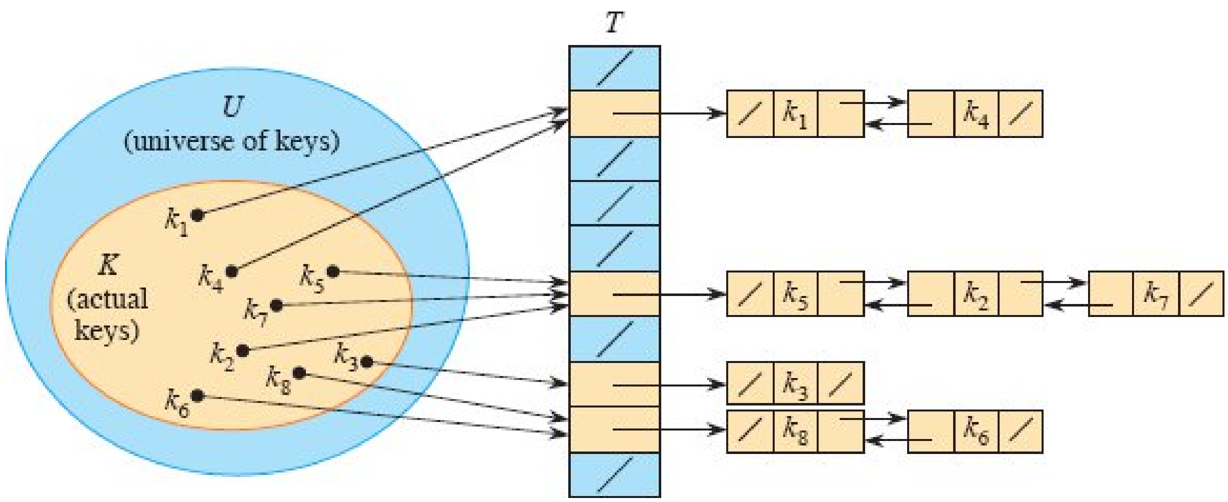

Figure 11.3 Collision resolution by chaining. Each nonempty hash-table slot $T[j]$ points to a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$. The list can be either singly or doubly linked. We show it as doubly linked because deletion may be faster that way when the deletion procedure knows which list element (not just which key) is to be deleted.

Collision resolution by chaining

At a high level, you can think of hashing with chaining as a nonrecursive form of divide-and-conquer: the input set of n elements is divided randomly into m subsets, each of approximate size n/m. A hash function determines which subset an element belongs to. Each subset is managed independently as a list.

Figure 11.3 shows the idea behind ***chaining***: each nonempty slot points to a linked list, and all the elements that hash to the same slot go into that slot's linked list. Slot j contains a pointer to the head of the list of all stored elements with hash value j. If there are no such elements, then slot j contains NIL.

When collisions are resolved by chaining, the dictionary operations are straightforward to implement. They appear on the next page and use the linked-list procedures from Section 10.2. The worst-case running time for insertion is $O(1)$. The insertion procedure is fast in part because it assumes that the element x being inserted is not already present in the table. To enforce this assumption, you can search (at additional cost) for an element whose key is $x.key$ before inserting. For searching, the worst-case running time is proportional to the length of the list. (We'll analyze this operation more closely below.) Deletion takes $O(1)$ time if the lists are doubly linked, as in Figure 11.3. (Since CHAINED-HASH-DELETE takes as input an element x and not its key k, no search is needed. If the hash table supports deletion, then its linked lists should be doubly linked in order to delete an item quickly. If the lists were only singly linked, then by Exercise 10.2-1, deletion could take time proportional to the length of the list. With singly linked lists, both deletion and searching would have the same asymptotic running times.)

```
CHAINED-HASH-INSERT(T, x)
LIST-PREPEND(T[h(x.key)], x)

CHAINED-HASH-SEARCH(T, k)
return LIST-SEARCH(T[h(k)], k)

CHAINED-HASH-DELETE(T, x)
LIST-DELETE(T[h(x.key)], x)
```

Analysis of hashing with chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given a hash table T with m slots that stores n elements, we define the ***load factor*** α for T as n/m, that is, the average number of elements stored in a chain. Our analysis will be in terms of α, which can be less than, equal to, or greater than 1.

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function—no better than using one linked list for all the elements. We clearly don't use hash tables for their worst-case performance.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average (meaning with respect to the distribution of keys to be hashed and with respect to the choice of hash function, if this choice is randomized). Section 11.3 discusses these issues, but for now we assume that any given element is equally likely to hash into any of the m slots. That is, the hash function is ***uniform***. We further assume that where a given element hashes to is *independent* of where any other elements hash to. In other words, we assume that we are using ***independent uniform hashing***.

Because hashes of distinct keys are assumed to be independent, independent uniform hashing is ***universal***: the chance that any two distinct keys k_1 and k_2 collide is at most $1/m$. Universality is important in our analysis and also in the specification of universal families of hash functions, which we'll see in Section 11.3.2.

For $j = 0, 1, \ldots, m - 1$, denote the length of the list $T[j]$ by n_j, so that

$$n = n_0 + n_1 + \cdots + n_{m-1} , \tag{11.1}$$

and the expected value of n_j is $E[n_j] = \alpha = n/m$.

We assume that $O(1)$ time suffices to compute the hash value $h(k)$, so that the time required to search for an element with key k depends linearly on the length $n_{h(k)}$ of the list $T[h(k)]$. Setting aside the $O(1)$ time required to compute the hash function and to access slot $h(k)$, we'll consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that the algorithm checks to see whether any have a key equal to k. We consider two cases. In the first, the search is unsuccessful: no element in the table has key k. In the second, the search successfully finds an element with key k.

Theorem 11.1

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes $\Theta(1 + \alpha)$ time on average, under the assumption of independent uniform hashing.

Proof Under the assumption of independent uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots. The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\Theta(1 + \alpha)$.

■

The situation for a successful search is slightly different. An unsuccessful search is equally likely to go to any slot of the hash table. A successful search, however, cannot go to an empty slot, since it is for an element that is present in one of the linked lists. We assume that the element searched for is equally likely to be any one of the elements in the table, so the longer the list, the more likely that the search is for one of its elements. Even so, the expected search time still turns out to be $\Theta(1 + \alpha)$.

Theorem 11.2
In a hash table in which collisions are resolved by chaining, a successful search takes $\Theta(1 + \alpha)$ time on average, under the assumption of independent uniform hashing.

Proof We assume that the element being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined during a successful search for an element x is 1 more than the number of elements that appear before x in x's list. Because new elements are placed at the front of the list, elements before x in the list were all inserted after x was inserted. Let x_i denote the ith element inserted into the table, for $i = 1, 2, \ldots, n$, and let $k_i = x_i.key$.

Our analysis uses indicator random variables extensively. For each slot q in the table and for each pair of distinct keys k_i and k_j, we define the indicator random variable

$X_{ijq} = \mathrm{I}\,\{\text{the search is for } x_i,\ h(k_i) = q, \text{ and } h(k_j) = q\}$.

That is, $X_{ijq} = 1$ when keys k_i and k_j collide at slot q and the search is for element x_i. Because Pr{the search is for x_i} = $1/n$, $\Pr\{h(k_i) = q\} = 1/m$, $\Pr\{h(k_j) = q\} = 1/m$, and these events are all independent, we have that $\Pr\{X_{ijq} = 1\} = 1/nm^2$. Lemma 5.1 on page 130 gives $E[X_{ijq}] = 1/nm^2$.

Next, we define, for each element x_j, the indicator random variable

$$\begin{aligned} Y_j &= \mathrm{I}\,\{x_j \text{ appears in a list prior to the element being searched for}\} \\ &= \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq} \ , \end{aligned}$$

,

since at most one of the X_{ijq} equals 1, namely when the element x_i being searched for belongs to the same list as x_j (pointed to by slot q), and $i < j$ (so that x_i appears after x_j in the list).

Our final random variable is Z, which counts how many elements appear in the list prior to the element being searched for:

$$Z = \sum_{j=1}^{n} Y_j \ .$$

Because we must count the element being searched for as well as all those preceding it in its list, we wish to compute $E[Z + 1]$. Using linearity of expectation (equation (C.24) on page 1192), we have

$$\begin{aligned} \mathrm{E}[Z+1] &= \mathrm{E}\left[1 + \sum_{j=1}^{n} Y_j\right] \\ &= 1 + \mathrm{E}\left[\sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \mathrm{E}\left[\sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} \mathrm{E}[X_{ijq}] \qquad \text{(by linearity of expectation)} \end{aligned}$$

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$.

■

What does this analysis mean? If the number of elements in the table is at most proportional to the number of hash-table slots, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked (assuming that the list element to be deleted is known, and not just its key), we can support all dictionary operations in $O(1)$ time on average.

The analysis in the preceding two theorems depends only on two essential properties of independent uniform hashing: uniformity (each key is equally likely to hash to any one of the m slots), and independence (so any two distinct keys collide with probability $1/m$).

Exercises

11.2-1

You use a hash function h to hash n distinct keys into an array T of length m. Assuming independent uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k_1, k_2\} : k_1 \neq k_2 \text{ and } h(k_1) = h(k_2)\}$?

11.2-2

Consider a hash table with 9 slots and the hash function $h(k) = k \bmod 9$. Demonstrate what happens upon inserting the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 with collisions resolved by chaining.

11.2-3

Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

11.2-4

Suggest how to allocate and deallocate storage for elements within the hash table itself by creating a "free list": a linked list of all the unused slots. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

11.2-5

You need to store a set of n keys in a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > (n-1)m$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

11.2-6

You have stored n keys in a hash table of size m, with collisions resolved by chaining, and you know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$.

11.3 Hash functions

For hashing to work well, it needs a good hash function. Along with being efficiently computable, what properties does a good hash function have? How do you design good hash functions?

This section first attempts to answer these questions based on two ad hoc approaches for creating hash functions: hashing by division and hashing by multiplication. Although these methods work well for some sets of input keys, they are limited because they try to provide a single fixed hash function that works well on any data—an approach called ***static hashing***.

We then see that provably good average-case performance for *any* data can be obtained by designing a suitable *family* of hash functions and choosing a hash function at random from this family at runtime, independent of the data to be hashed. The approach we examine is called random hashing. A particular kind of random hashing, universal hashing, works well. As we saw with quicksort in Chapter 7, randomization is a powerful algorithmic design tool.

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of independent uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other keys have hashed to. What does "equally likely" mean here? If the hash function is fixed, any probabilities would have to be based on the probability distribution of the input keys.

Unfortunately, you typically have no way to check this condition, unless you happen to know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

Occasionally you might know the distribution. For example, if you know that the keys are random real numbers k independently and uniformly distributed in the range $0 \le k < 1$, then the hash function

$$h(k) = \lfloor km \rfloor$$

satisfies the condition of independent uniform hashing.

A good static hashing approach derives the hash value in a way that you expect to be independent of any patterns that might exist in the data. For example, the "division method" (discussed in Section 11.3.1) computes the hash value as the remainder when the key is divided by a specified prime number. This method may give good results, if you (somehow) choose a prime number that is unrelated to any patterns in the distribution of keys.

Random hashing, described in Section 11.3.2, picks the hash function to be used at random from a suitable family of hashing functions. This approach removes any need to know anything about the probability distribution of the input keys, as the randomization necessary for good average-case behavior then comes from the (known) random process used to pick the hash function from the family of hash functions, rather than from the (unknown) process used to create the input keys. We recommend that you use random hashing.

Keys are integers, vectors, or strings

In practice, a hash function is designed to handle keys that are one of the following two types:

- A short nonnegative integer that fits in a w-bit machine word. Typical values for w would be 32 or 64.
- A short vector of nonnegative integers, each of bounded size. For example, each element might be an 8-bit byte, in which case the vector is often called a (byte) string. The vector might be of variable length.

To begin, we assume that keys are short nonnegative integers. Handling vector keys is more complicated and discussed in Sections 11.3.5 and 11.5.2.

11.3.1 Static hashing

Static hashing uses a single, fixed hash function. The only randomization available is through the (usually unknown) distribution of input keys. This section discusses two standard approaches for static hashing: the division method and the multiplication method. Although static hashing is no longer recommended, the multiplication method also provides a good foundation for "nonstatic" hashing—better known as random hashing—where the hash function is chosen at random from a suitable family of hash functions.

The division method

The ***division method*** for creating hash functions maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

The division method may work well when m is a prime not too close to an exact power of 2. There is no guarantee that this method provides good average-case performance, however, and it may complicate applications since it constrains the size of the hash tables to be prime.

The multiplication method

The general ***multiplication method*** for creating hash functions operates in two steps. First, multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, multiply this value by m and take the floor of the result. That is, the hash function is

$$h(k) = \lfloor m\,(kA \bmod 1)\rfloor,$$

where "kA mod 1" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$. The general multiplication method has the advantage that the value of m is not critical and you can choose it independently of how you choose the multiplicative constant A.

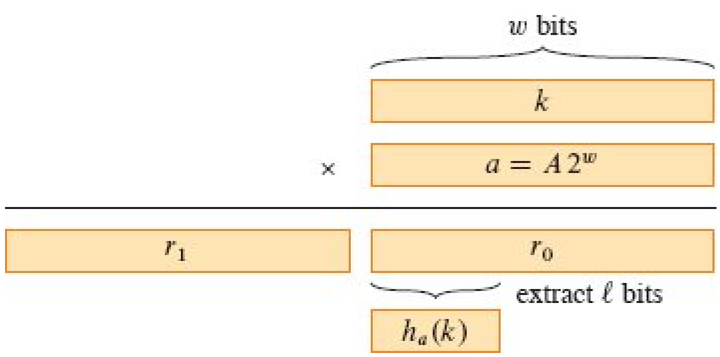

Figure 11.4 The multiply-shift method to compute a hash function. The w-bit representation of the key k is multiplied by the w-bit value $a = A \cdot 2^w$. The ℓ highest-order bits of the lower w-bit half of the product form the desired hash value $h_a(k)$.

The multiply-shift method

In practice, the multiplication method is best in the special case where the number m of hash-table slots is an exact power of 2, so that $m = 2^\ell$ for some integer ℓ, where $\ell \le w$ and w is the number of bits in a machine word. If you choose a fixed w-bit positive integer $a = A\,2^w$, where $0 < A < 1$ as in the multiplication method so that a is in the range $0 < a < 2^w$, you can implement the function on most computers as follows. We assume that a key k fits into a single w-bit word.

Referring to Figure 11.4, first multiply k by the w-bit integer a. The result is a $2w$-bit value $r_1 2^w + r_0$, where r_1 is the high-order w-bit word of the product and r_0 is the low-order w-bit word of the product. The desired ℓ-bit hash value consists of the ℓ most significant bits of r_0. (Since r_1 is ignored, the hash function can be implemented on a computer that produces only a w-bit product given two w-bit inputs, that is, where the multiplication operation computes modulo 2^w.)

In other words, you define the hash function $h = h_a$, where

$$h_a(k) = (ka \bmod 2^w) \ggg (w - \ell) \tag{11.2}$$

for a fixed nonzero w-bit value a. Since the product ka of two w-bit words occupies $2w$ bits, taking this product modulo 2^w zeroes out the high-order w bits (r_1), leaving only the low-order w bits (r_0). The $\ggg$ operator performs a logical right shift by $w - \ell$ bits, shifting zeros into the vacated positions on the left, so that the ℓ most significant bits of r_0 move into the ℓ rightmost positions. (It's the same as dividing by $2^{w-\ell}$ and taking the floor of the result.) The resulting value equals the ℓ most significant bits of r_0. The hash function h_a can be implemented with three machine instructions: multiplication, subtraction, and logical right shift.

As an example, suppose that $k = 123456$, $\ell = 14$, $m = 2^{14} = 16384$, and $w = 32$. Suppose further that we choose $a = 2654435769$ (following a suggestion of Knuth [261]). Then $ka = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, and so $r_1 = 76300$ and $r_0 = 17612864$. The 14 most significant bits of r_0 yield the value $h_a(k) = 67$.

Even though the multiply-shift method is fast, it doesn't provide any guarantee of good average-case performance. The universal hashing approach presented in the next section provides such a guarantee. A simple randomized variant of the multiply-shift method works well on the average, when the program begins by picking a as a randomly chosen odd integer.

11.3.2 Random hashing

Suppose that a malicious adversary chooses the keys to be hashed by some fixed hash function. Then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Any static hash function is vulnerable to such terrible worst-case behavior. The only effective way to improve the situation is to choose the hash function *randomly* in a way that is *independent* of the keys that are actually going to be stored. This approach is called ***random hashing***. A special case of this approach, called ***universal hashing***, can yield provably good performance on average when collisions are handled by chaining, no matter which keys the adversary chooses.

To use random hashing, at the beginning of program execution you select the hash function at random from a suitable family of functions. As in the case of quicksort, randomization guarantees that no single input always evokes worst-case behavior. Because you randomly select the hash function, the algorithm can behave differently on each execution, even for the same set of keys to be hashed, guaranteeing good average-case performance.

Let $\mathscr{H}$ be a finite family of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m - 1\}$. Such a family is said to be ***universal*** if for each pair of distinct keys $k_1, k_2 \in U$, the number of hash functions $h \in \mathscr{H}$ for which $h(k_1) = h(k_2)$ is at most $|\mathscr{H}|/m$. In other words, with a hash function randomly chosen from $\mathscr{H}$, the chance of a collision between distinct keys k_1 and k_2 is no more than the chance $1/m$ of a collision if $h(k_1)$ and $h(k_2)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m - 1\}$.

Independent uniform hashing is the same as picking a hash function uniformly at random from a family of m^n hash functions, each member of that family mapping the n keys to the m hash values in a different way.

Every independent uniform random family of hash function is universal, but the converse need not be true: consider the case where $U = \{0, 1, \ldots, m - 1\}$ and the only hash function in the family is the identity function. The probability that two distinct keys collide is zero, even though each key is hashes to a fixed value.

The following corollary to Theorem 11.2 on page 279 says that universal hashing provides the desired payoff: it becomes impossible for an adversary to pick a sequence of operations that forces the worst-case running time.

Corollary 11.3

Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes $\Theta(s)$ expected time to handle any sequence of s INSERT, SEARCH, and DELETE operations containing $n = O(m)$ INSERT operations.

Proof The INSERT and DELETE operations take constant time. Since the number n of insertions is $O(m)$, we have that $\alpha = O(1)$. Furthermore, the expected time for each SEARCH operation is $O(1)$, which can be seen by examining the proof of Theorem 11.2. That analysis depends only on collision probabilities, which are $1/m$ for any pair k_1, k_2 of keys by the choice of an independent uniform hash function in that theorem. Using a universal family of hash functions here instead of using independent uniform hashing changes the probability of collision from $1/m$ to at most $1/m$. By linearity of expectation, therefore, the expected time for the entire sequence of s operations is $O(s)$. Since each operation takes $\Omega(1)$ time, the $\Theta(s)$ bound follows.

■

11.3.3 Achievable properties of random hashing

There is a rich literature on the properties a family $\mathscr{H}$ of hash functions can have, and how they relate to the efficiency of hashing. We summarize a few of the most interesting ones here.

Let $\mathscr{H}$ be a family of hash functions, each with domain U and range $\{0, 1, \ldots, m-1\}$, and let h be any hash function that is picked uniformly at random from $\mathscr{H}$. The probabilities mentioned are probabilities over the picks of h.

- The family $\mathscr{H}$ is ***uniform*** if for any key k in U and any slot q in the range $\{0, 1, \ldots, m-1\}$, the probability that $h(k) = q$ is $1/m$.
- The family $\mathscr{H}$ is ***universal*** if for any distinct keys k_1 and k_2 in U, the probability that $h(k_1) = h(k_2)$ is at most $1/m$.
- The family $\mathscr{H}$ of hash functions is ***ϵ-universal*** if for any distinct keys k_1 and k_2 in U, the probability that $h(k_1) = h(k_2)$ is at most ϵ. Therefore, a universal family of hash functions is also $1/m$-universal.[2]
- The family $\mathscr{H}$ is ***d-independent*** if for any distinct keys $k_1, k_2, \ldots, k_d$ in U and any slots $q_1, q_2, \ldots, q_d$, not necessarily distinct, in $\{0, 1, \ldots, m-1\}$ the probability that $h(k_i) = q_i$ for $i = 1, 2, \ldots, d$ is $1/m^d$.

Universal hash-function families are of particular interest, as they are the simplest type supporting provably efficient hash-table operations for any input data set. Many other interesting and desirable properties, such as those noted above, are also possible and allow for efficient specialized hash-table operations.

11.3.4 Designing a universal family of hash functions

This section present two ways to design a universal (or ϵ-universal) family of hash functions: one based on number theory and another based on a randomized variant of the multiply-shift method presented in Section 11.3.1. The first method is a bit easier to prove universal, but the second method is newer and faster in practice.

A universal family of hash functions based on number theory

We can design a universal family of hash functions using a little number theory. You may wish to refer to Chapter 31 if you are unfamiliar with basic concepts in number theory.

Begin by choosing a prime number p large enough so that every possible key k lies in the range 0 to $p - 1$, inclusive. We assume here that p has a "reasonable" length. (See Section 11.3.5 for a discussion of methods for handling long input keys, such as variable-length strings.) Let $\mathbb{Z}_p$ denote the set $\{0, 1, \ldots, p - 1\}$, and let $\mathbb{Z}_p^*$ denote the set $\{1, 2, \ldots, p - 1\}$. Since p is prime, we can solve equations modulo p with the methods given in Chapter 31. Because the size of the universe of keys is greater than the number of slots in the hash table (otherwise, just use direct addressing), we have $p > m$.

Given any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$, define the hash function h_{ab} as a linear transformation followed by reductions modulo p and then modulo m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m \; . \tag{11.3}$$

For example, with $p = 17$ and $m = 6$, we have

$$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5. \end{aligned}$$

Given p and m, the family of all such hash functions is

$$\mathcal{H}_{pm} = \left\{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\right\} \; . \tag{11.4}$$

Each hash function h_{ab} maps $\mathbb{Z}_p$ to $\mathbb{Z}_m$. This family of hash functions has the nice property that the size m of the output range (which is the size of the hash table) is arbitrary—it need not be prime. Since you can choose from among $p - 1$ values for a and p values for b, the family $\mathcal{H}_{pm}$ contains $p(p - 1)$ hash functions.

Theorem 11.4

The family $\mathscr{H}_{pm}$ of hash functions defined by equations (11.3) and (11.4) is universal.

Proof Consider two distinct keys k_1 and k_2 from $\mathbb{Z}_p$, so that $k_1 \neq k_2$. For a given hash function h_{ab}, let

$$r_1 = (ak_1 + b) \bmod p,$$

$$r_2 = (ak_2 + b) \bmod p.$$

We first note that $r_1 \neq r_2$. Why? Since we have $r_1 - r_2 = a(k_1 - k_2) \pmod p$, it follows that $r_1 \neq r_2$ because p is prime and both a and $(k_1 - k_2)$ are nonzero modulo p. By Theorem 31.6 on page 908, their product must also be nonzero modulo p. Therefore, when computing any $h_{ab} \in \mathscr{H}_{pm}$, distinct inputs k_1 and k_2 map to distinct values r_1 and r_2 modulo p, and there are no collisions yet at the "mod p level." Moreover, each of the possible $p(p - 1)$ choices for the pair (a, b) with $a \neq 0$ yields a *different* resulting pair (r_1, r_2) with $r_1 \neq r_2$, since we can solve for a and b given r_1 and r_2:

$$a = ((r - r_2)((k_1 - k_2)^{-1} \bmod p)) \bmod p,$$

$$b = (r_1 - ak_1) \bmod p,$$

where $((k_1 - k_2)^{-1} \bmod p)$ denotes the unique multiplicative inverse, modulo p, of $k_1 - k_2$. For each of the p possible values of r_1, there are only $p - 1$ possible values of r_2 that do not equal r_1, making only $p(p - 1)$ possible pairs (r_1, r_2) with $r_1 \neq r_2$. Therefore, there is a one-to-one correspondence between pairs (a, b) with $a \neq 0$ and pairs (r_1, r_2) with $r_1 \neq r_2$. Thus, for any given pair of distinct inputs k_1 and k_2, if we pick (a, b) uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$, the resulting pair (r_1, r_2) is equally likely to be any pair of distinct values modulo p.

Therefore, the probability that distinct keys k_1 and k_2 collide is equal to the probability that $r_1 = r_2 \pmod m$ when r_1 and r_2 are randomly chosen as distinct values modulo p. For a given value of r_1, of the $p - 1$ possible remaining values for r_2, the number of values r_2 such that $r_2 \neq r_1$ and $r_2 = r_1 \pmod m$ is at most

$$\begin{aligned} \left\lceil \frac{p}{m} \right\rceil - 1 &\leq \frac{p+m-1}{m} - 1 \quad \text{(by inequality (3.7) on page 64)} \\ &= \frac{p-1}{m} . \end{aligned}$$

The probability that r_2 collides with r_1 when reduced modulo m is at most $((p - 1)/m)/(p - 1) = 1/m$, since r_2 is equally likely to be any of the $p - 1$ values in Z_p that are different from r_1, but at most $(p - 1)/m$ of those values are equivalent to r_1 modulo m.

Therefore, for any pair of distinct values $k_1, k_2 \in \mathbb{Z}_p$,

$$\Pr\{h_{ab}(k_1) = h_{ab}(k_2)\} \leq 1/m,$$

so that $\mathscr{H}_{pm}$ is indeed universal.

■

A 2/*m*-universal family of hash functions based on the multiply-shift method

We recommend that in practice you use the following hash-function family based on the multiply-shift method. It is exceptionally efficient and (although we omit the proof) provably $2/m$-universal. Define $\mathscr{H}$ to be the family of multiply-shift hash functions with odd constants a:

$$\mathscr{H} = \{h_a : a \text{ is odd}, 1 \leq a < m, \text{and } h_a \text{ is defined by equation (11.2)}\} . \qquad (11.5)$$

Theorem 11.5

The family of hash functions $\mathscr{H}$ given by equation (11.5) is $2/m$-universal.

■

That is, the probability that any two distinct keys collide is at most $2/m$. In many practical situations, the speed of computing the hash function more than compensates for the higher upper bound on the probability that two distinct keys collide when compared with a universal hash function.

11.3.5 Hashing long inputs such as vectors or strings

Sometimes hash function inputs are so long that they cannot be easily encoded modulo a reasonably sized prime number p or encoded within a single word of, say, 64 bits. As an example, consider the class of vectors, such as vectors of 8-bit bytes (which is how strings in many programming languages are stored). A vector might have an arbitrary nonnegative length, in which case the length of the input to the hash function may vary from input to input.

Number-theoretic approaches

One way to design good hash functions for variable-length inputs is to extend the ideas used in Section 11.3.4 to design universal hash functions. Exercise 11.3-6 explores one such approach.

Cryptographic hashing

Another way to design a good hash function for variable-length inputs is to use a hash function designed for cryptographic applications. ***Cryptographic hash functions*** are complex pseudorandom functions, designed for applications requiring properties beyond those needed here, but are robust, widely implemented, and usable as hash functions for hash tables.

A cryptographic hash function takes as input an arbitrary byte string and returns a fixed-length output. For example, the NIST standard deterministic cryptographic hash function SHA-256 [346] produces a 256-bit (32-byte) output for any input.

Some chip manufacturers include instructions in their CPU architectures to provide fast implementations of some cryptographic functions. Of particular interest are instructions that efficiently implement rounds of the Advanced Encryption Standard (AES), the “AES-NI” instructions. These instructions execute in a few tens of nanoseconds, which is generally fast enough for use with hash tables. A message authentication code such as CBC-MAC based on AES and the use of the AES-NI instructions could be a useful and efficient hash function. We don't pursue the potential use of specialized instruction sets further here.

Cryptographic hash functions are useful because they provide a way of implementing an approximate version of a random oracle. As noted earlier, a random oracle is equivalent to an independent uniform hash function family. From a theoretical point of view, a random oracle is an unachievable ideal: a deterministic function that provides a randomly selected output for each input. Because it is deterministic, it provides the same output if queried again for the same input. From a practical point of view, constructions of hash function families based on cryptographic hash functions are sensible substitutes for random oracles.

There are many ways to use a cryptographic hash function as a hash function. For example, we could define

$$h(k) = \text{SHA-256}(k) \bmod m.$$

To define a family of such hash functions one may prepend a "salt" string a to the input before hashing it, as in

$$h_a(k) = \text{SHA-256}(a \parallel k) \bmod m,$$

where $a \parallel k$ denotes the string formed by concatenating the strings a and k. The literature on message authentication codes (MACs) provides additional approaches.

Cryptographic approaches to hash-function design are becoming more practical as computers arrange their memories in hierarchies of differing capacities and speeds. Section 11.5 discusses one hash-function design based on the RC6 encryption method.

Exercises

11.3-1

You wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character string.

How might you take advantage of the hash values when searching the list for an element with a given key?

11.3-2

You hash a string of r characters into m slots by treating it as a radix-128 number and then using the division method. You can represent the number m as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can you apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

11.3-3

Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p. Show that if string x can be converted to string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

11.3-4

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m\ (kA \bmod 1)\rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

★ *11.3-5*

Show that any ϵ-universal family $\mathscr{H}$ of hash functions from a finite set U to a finite set Q has $\epsilon \geq 1/|Q| - 1/|U|$.

★ *11.3-6*

Let U be the set of d-tuples of values drawn from $\mathbb{Z}_p$, and let $Q = \mathbb{Z}_p$, where p is prime. Define the hash function $h_b : U \to Q$ for $b \in \mathbb{Z}_p$ on an input d-tuple $\langle a_0, a_1, \ldots, a_{d-1}\rangle$ from U as

$$h_b(\langle a_0, a_1, \ldots, a_{d-1}\rangle) = \left(\sum_{j=0}^{d-1} a_j b^j\right) \bmod p ,$$

and let $\mathscr{H} = \{h_b : b \in \mathbb{Z}_p\}$. Argue that $\mathscr{H}$ is ϵ-universal for $\epsilon = (d - 1)/p$. (*Hint:* See Exercise 31.4-4.)

11.4 Open addressing

This section describes open addressing, a method for collision resolution that, unlike chaining, does not make use of storage outside of the hash table itself. In ***open addressing***, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. No lists or elements are stored outside the table, unlike in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made. One consequence is that the load factor α can never exceed 1.

Collisions are handled as follows: when a new element is to be inserted into the table, it is placed in its "first-choice" location if possible. If that location is already occupied, the new element is placed in its "second-choice" location. The process continues until an empty slot is found in which to place the new element. Different elements have different preference orders for the locations.

To search for an element, systematically examine the preferred table slots for that element, in order of decreasing preference, until either you find the desired element or you find an empty slot and thus verify that the element is not in the table.

Of course, you could use chaining and store the linked lists inside the hash table, in the otherwise unused hash-table slots (see Exercise 11.2-4), but the advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, you compute the sequence of slots to be examined. The memory freed by not storing pointers provides the hash table with a larger number of slots in the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, successively examine, or ***probe***, the hash table until you find an empty slot in which to put the key. Instead of being fixed in the order 0, 1, …, $m - 1$ (which implies a $\Theta(n)$ search time), the sequence of positions probed depends upon the key being inserted. To determine which slots to probe, the hash function includes the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m - 1\} \to \{0, 1, \ldots, m - 1\}.$$

Open addressing requires that for every key k, the ***probe sequence*** $\langle h(k, 0), h(k, 1), \ldots, h(k, m - 1)\rangle$ be a permutation of $\langle 0, 1, \ldots, m - 1\rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. The HASH-INSERT procedure on the following page assumes that the elements in the hash table T are keys with no satellite information: the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty). The HASH-INSERT procedure takes as input a hash table T and a key k that is assumed to be not already present in the hash table. It either returns the slot number where it stores key k or flags an error because the hash table is already full.

```
HASH-INSERT(T, k)
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error "hash table overflow"
```

```
HASH-SEARCH(T, k)
i = 0
repeat
```

```
    q = h(k, i)
    if T[q] == k
        return q
    i = i + 1
until T[q] == NIL or i == m
return NIL
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence. The procedure HASH-SEARCH takes as input a hash table T and a key k, returning q if it finds that slot q contains key k, or NIL if key k is not present in table T.

Deletion from an open-address hash table is tricky. When you delete a key from slot q, it would be a mistake to mark that slot as empty by simply storing NIL in it. If you did, you might be unable to retrieve any key k for which slot q was probed and found occupied when k was inserted. One way to solve this problem is by marking the slot, storing in it the special value DELETED instead of NIL. The HASH-INSERT procedure then has to treat such a slot as empty so that it can insert a new key there. The HASH-SEARCH procedure passes over DELETED values while searching, since slots containing DELETED were filled when the key being searched for was inserted. Using the special value DELETED, however, means that search times no longer depend on the load factor α, and for this reason chaining is frequently selected as a collision resolution technique when keys must be deleted. There is a simple special case of open addressing, linear probing, that avoids the need to mark slots with DELETED. Section 11.5.1 shows how to delete from a hash table when using linear probing.

In our analysis, we assume ***independent uniform permutation hashing*** (also confusingly known as ***uniform hashing*** in the literature): the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, \ldots, m - 1\rangle$. Independent uniform permutation hashing generalizes the notion of independent uniform hashing defined earlier to a hash function that produces not just a single slot number, but a whole probe sequence. True independent uniform permutation hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used.

We'll examine both double hashing and its special case, linear probing. These techniques guarantee that $\langle h(k, 0), h(k, 1), \ldots, h(k, m - 1)\rangle$ is a permutation of $\langle 0, 1, \ldots, m - 1\rangle$ for each key k. (Recall that the second parameter to the hash function h is the probe number.) Neither double hashing nor linear probing meets the assumption of independent uniform permutation hashing, however. Double hashing cannot generate more than m^2 different probe sequences (instead of the $m!$ that independent uniform permutation hashing requires). Nonetheless, double hashing has a large number of possible probe sequences and, as you might expect, seems to give good results. Linear probing is even more restricted, capable of generating only m different probe sequences.

Double hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. ***Double hashing*** uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

where both h_1 and h_2 are ***auxiliary hash functions***. The initial probe goes to position $T[h_1(k)]$, and successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m. Thus, the probe sequence here depends in two ways upon the key k, since the initial probe position $h_1(k)$, the step size $h_2(k)$, or both, may vary. Figure 11.5 gives an example of insertion by double hashing.

In order for the entire hash table to be searched, the value $h_2(k)$ must be relatively prime to the hash-table size m. (See Exercise 11.4-5.) A convenient way to ensure this condition is to let m be an exact power of 2 and to design h_2 so that it always produces an odd number. Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m. For example, you could choose m prime and let

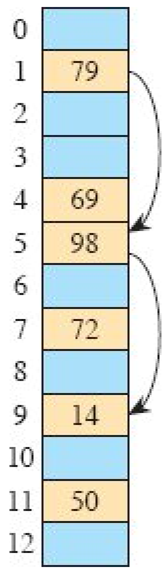

Figure 11.5 Insertion by double hashing. The hash table has size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 = 1 \pmod{13}$ and $14 = 3 \pmod{11}$, the key 14 goes into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

$$h_1(k) = k \bmod m,$$

$$h_2(k) = 1 + (k \bmod m'),$$

where m' is chosen to be slightly less than m (say, $m - 1$). For example, if $k = 123456$, $m = 701$, and $m' = 700$, then $h_1(k) = 80$ and $h_2(k) = 257$, so that the first probe goes to position 80, and successive probes examine every 257th slot (modulo m) until the key has been found or every slot has been examined.

Although values of m other than primes or exact powers of 2 can in principle be used with double hashing, in practice it becomes more difficult to efficiently generate $h_2(k)$ (other than choosing $h_2(k) = 1$, which gives linear probing) in a way that ensures that it is relatively prime to m, in part because the relative density $\phi(m)/m$ of such numbers for general m may be small (see equation (31.25) on page 921).

When m is prime or an exact power of 2, double hashing produces $\Theta(m^2)$ probe sequences, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence. As a result, for such values of m, double hashing appears to perform close to the "ideal" scheme of independent uniform permutation hashing.

Linear probing

Linear probing, a special case of double hashing, is the simplest open-addressing approach to resolving collisions. As with double hashing, an auxiliary hash function h_1 determines the first probe position $h_1(k)$ for inserting an element. If slot $T[h_1(k)]$ is already occupied, probe the next position $T[h_1(k) + 1]$. Keep going as necessary, on up to slot $T[m - 1]$, and then wrap around to slots $T[0]$, $T[1]$, and so on, but never going past slot $T[h_1(k) - 1]$. To view linear probing as a special case of double hashing, just set the double-hashing step function h_2 to be fixed at 1: $h_2(k) = 1$ for all k. That is, the hash function is

$$h(k, i) = (h_1(k) + i) \bmod m \tag{11.6}$$

for $i = 0, 1, \ldots, m - 1$. The value of $h_1(k)$ determines the entire probe sequence, and so assuming that $h_1(k)$ can take on any value in $\{0, 1, \ldots, m - 1\}$, linear probing allows only m distinct probe sequences.

We'll revisit linear probing in Section 11.5.1.

Analysis of open-address hashing

As in our analysis of chaining in Section 11.2, we analyze open addressing in terms of the load factor $\alpha = n/m$ of the hash table. With open addressing, at most one element occupies each slot, and thus $n \le m$, which implies $\alpha \le 1$. The analysis below requires α to be strictly less than 1, and so we assume that at least one slot is empty. Because deleting from an open-address hash table does not really free up a slot, we assume as well that no deletions occur.

For the hash function, we assume independent uniform permutation hashing. In this idealized scheme, the probe sequence $\langle h(k, 0), h(k, 1),$

…, $h(k, m-1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0, 1, \ldots, m-1\rangle$. Of course, any given key has a unique fixed probe sequence associated with it. What we mean here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyze the expected number of probes for hashing with open addressing under the assumption of independent uniform permutation hashing, beginning with the expected number of probes made in an unsuccessful search (assuming, as stated above, that $\alpha < 1$).

The bound proven, of $1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$, has an intuitive interpretation. The first probe always occurs. With probability approximately α, the first probe finds an occupied slot, so that a second probe happens. With probability approximately α^2, the first two slots are occupied so that a third probe ensues, and so on.

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming independent uniform permutation hashing and no deletions.

Proof In an unsuccessful search, every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty. Let the random variable X denote the number of probes made in an unsuccessful search, and define the event A_i, for $i = 1, 2, \ldots$, as the event that an ith probe occurs and it is to an occupied slot. Then the event $\{X \geq i\}$ is the intersection of events $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$. We bound $\Pr\{X \geq i\}$ by bounding $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$. By Exercise C.2-5 on page 1190,

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}.$$