

Pointer-based data structures also seem to be a folk invention. According to Knuth, pointers were apparently used in early computers with drum memories. The A-1 language developed by G. M. Hopper in 1951 represented algebraic formulas as binary trees. Knuth credits the IPL-II language, developed in 1956 by A. Newell, J. C. Shaw, and H. A. Simon, for recognizing the importance and promoting the use of pointers. Their IPL-III language, developed in 1957, included explicit stack operations.

Chapter 11: Hash Tables

Overview

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler for a computer language maintains a symbol table, in which the keys of elements are arbitrary character strings that correspond 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 expected time to search for an element in a hash table is $O(1)$.

A hash table is a generalization of the simpler notion of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in $O(1)$ time. [Section 11.1](#) discusses direct addressing in more detail. Direct addressing is applicable when we can afford to allocate an array that has one 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, the array index is *computed* 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 array indices can be computed 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](#) explains how "perfect hashing" can support searches in $O(1)$ *worst-case* time, when the set of keys being stored is static (that is, when the set of keys never changes once stored).

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 key drawn from the universe $U = \{0, 1, \dots, m - 1\}$, where m is not too large. We shall assume that no two elements have the same key.

To represent the dynamic set, we use an array, or *direct-address table*, denoted by $T[0 \dots m - 1]$, in which each position, or *slot*, corresponds to a key in the universe U . [Figure 11.1](#)

illustrates the approach; slot k points to an element in the set with key k . If the set contains no element with key k , then $T[k] = \text{NIL}$.

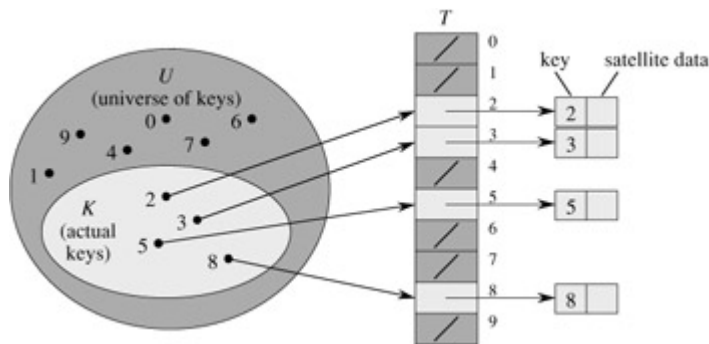


Figure 11.1: Implementing a dynamic set by a direct-address table T . Each key in the universe $U = \{0, 1, \dots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,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

The dictionary operations are trivial to implement.

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

```
DIRECT-ADDRESS-INSERT( $T, x$ )
     $T[\text{key}[x]] \leftarrow x$ 
```

```
DIRECT-ADDRESS-DELETE( $T, x$ )
     $T[\text{key}[x]] \leftarrow \text{NIL}$ 
```

Each of these operations is fast: only $O(1)$ time is required.

For some applications, the elements in the dynamic set can be stored in the direct-address table itself. 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, we can store the object in the slot itself, thus saving space. Moreover, it is often unnecessary to store the key field of the object, since if we have the index of an object in the table, we have its key. If keys are not stored, however, we must have some way to tell if the slot is empty.

Exercises 11.1-1

Suppose that 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?

Exercises 11.1-2

A **bit vector** is simply an array of bits (0's and 1's). 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 with no Satellite Data. Dictionary Operations Should Run in $O(1)$ Time.

Exercises 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.)

Exercises 11.1-4: ★

We wish to implement a dictionary by using direct addressing on a *huge* array. 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 the initialization of the data structure should take $O(1)$ time. (*Hint*: Use an additional 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 difficulty with direct addressing is obvious: if the universe U is large, 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 requirements can be reduced to $\Theta(|K|)$ while we maintain the benefit that searching for an element in the hash table still requires only $O(1)$ time. The only catch is that this bound is for the *average time*, whereas for direct addressing it holds for the *worst-case time*.

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

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

We say that an element with key k **hashes** to slot $h(k)$; we also say that $h(k)$ is the **hash value** of key k . [Figure 11.2](#) illustrates the basic idea. The point of the hash function is to reduce the range of array indices that need to be handled. Instead of $|U|$ values, we need to handle only m values. Storage requirements are correspondingly reduced.

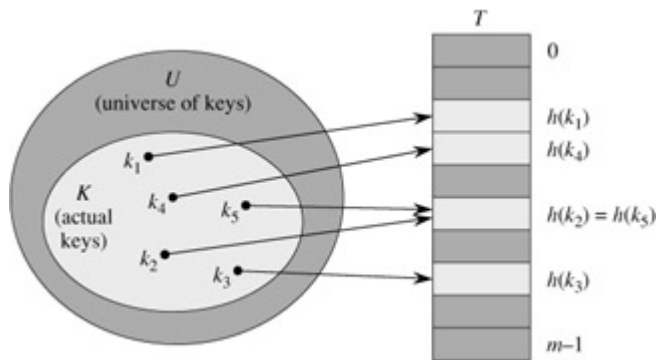


Figure 11.2: Using a hash function h to map keys to hash-table slots. keys k_2 and k_5 map to the same slot, so they collide.

There is one hitch: 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 would be 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 should always produce the same output $h(k)$.) Since $|U| > m$, however, there must be at least two keys that have the same hash value; avoiding collisions altogether is therefore impossible. Thus, while a well-designed, "random"-looking hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur.

The remainder of this section presents the simplest collision resolution technique, called chaining. [Section 11.4](#) introduces an alternative method for resolving collisions, called open addressing.

Collision resolution by chaining

In **chaining**, we put all the elements that hash to the same slot in a linked list, as shown in [Figure 11.3](#). Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL.

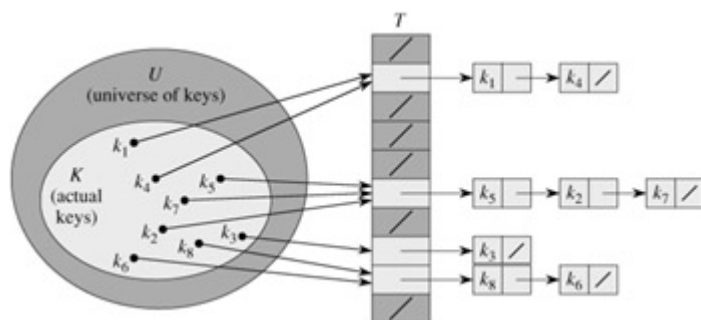


Figure 11.3: Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j . For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

The dictionary operations on a hash table T are easy to implement when collisions are resolved by chaining.

```

CHAINED-HASH-INSERT( $T, x$ )
    insert  $x$  at the head of list  $T[h(\text{key}[x])]$ 

CHAINED-HASH-SEARCH( $T, k$ )
    search for an element with key  $k$  in list  $T[h(k)]$ 

CHAINED-HASH-DELETE( $T, x$ )
    delete  $x$  from the list  $T[h(\text{key}[x])]$ 

```

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; this assumption can be checked if necessary (at additional cost) by performing a search before insertion. For searching, the worst-case running time is proportional to the length of the list; we shall analyze this operation more closely below. Deletion of an element x can be accomplished in $O(1)$ time if the lists are doubly linked. (Note that CHAINED-HASH-DELETE takes as input an element x and not its key k , so we don't have to search for x first. If the lists were singly linked, it would not be of great help to take as input the element x rather than the key k . We would still have to find x in the list $T[h(\text{key}[x])]$, so that the *next* link of x 's predecessor could be properly set to splice x out. In this case, deletion and searching would have essentially the same running time.)

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 if we used one linked list for all the elements. Clearly, hash tables are not used for their worst-case performance. (Perfect hashing, described in [Section 11.5](#), does however provide good worst-case performance when the set of keys is static.)

The average 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. [Section 11.3](#) discusses these issues, but for now we shall assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this the assumption of **simple uniform hashing**.

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

$$(11.1) \quad n = n_0 + n_1 + \dots + n_{m-1},$$

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

We assume that the hash value $h(k)$ can be computed in $O(1)$ time, 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 $h(k)$, let us

consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that are checked to see if their keys are equal to k . We shall 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 expected time $\Theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Proof Under the assumption of simple 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, since each list is not equally likely to be searched. Instead, the probability that a list is searched is proportional to the number of elements it contains. Nonetheless, the expected search time is still $\Theta(1 + \alpha)$.

Theorem 11.2

In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple 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. Elements before x in the list were all inserted after x was inserted, because new elements are placed at the front of the list. To find the expected number of elements examined, we take the average, over the n elements x in the table, of 1 plus the expected number of elements added to x 's list after x was added to the list. Let x_i denote the i th element inserted into the table, for $i = 1, 2, \dots, n$, and let $k_i = \text{key}[x_i]$. For keys k_i and k_j , we define the indicator random variable $X_{ij} = I\{h(k_i) = h(k_j)\}$. Under the assumption of simple uniform hashing, we have $\Pr\{h(k_i) = h(k_j)\} = 1/m$, and so by [Lemma 5.1](#), $E[X_{ij}] = 1/m$. Thus, the expected number of elements examined in a successful search is

$$\begin{aligned}
& \mathbb{E} \left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n X_{ij} \right) \right] \\
&= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \right) \quad (\text{by linearity of expectation}) \\
&= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m} \right) \\
&= 1 + \frac{1}{nm} \sum_{i=1}^n (n-i) \\
&= 1 + \frac{1}{nm} \left(\sum_{i=1}^n n - \sum_{i=1}^n i \right) \\
&= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2} \right) \quad (\text{by equation (A.1)}) \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.
\end{aligned}$$

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 hash-table slots is at least proportional to the number of elements in the table, 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, all dictionary operations can be supported in $O(1)$ time on average.

Exercises 11.2-1

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

Exercises 11.2-2

Demonstrate the insertion of the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

Exercises 11.2-3

Professor Marley hypothesizes that substantial performance gains can be obtained if we modify the chaining scheme so that each list is kept in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

Exercises 11.2-4

Suggest how storage for elements can be allocated and deallocated within the hash table itself by linking all unused slots into a free list. 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?

Exercises 11.2-5

Show that if $|U| > nm$, there is a subset of U 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 $\Theta(n)$.

11.3 Hash functions

In this section, we discuss some issues regarding the design of good hash functions and then present three schemes for their creation. Two of the schemes, hashing by division and hashing by multiplication, are heuristic in nature, whereas the third scheme, universal hashing, uses randomization to provide provably good performance.

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to. Unfortunately, it is typically not possible to check this condition, since one rarely knows the probability distribution according to which the keys are drawn, and the keys may not be drawn independently.

Occasionally we do know the distribution. For example, if the keys are known to be random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$, the hash function

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

satisfies the condition of simple uniform hashing.

In practice, heuristic techniques can often be used to create a hash function that performs well. Qualitative information about distribution of keys may be useful in this design process. For example, consider a compiler's symbol table, in which the keys are character strings representing identifiers in a program. Closely related symbols, such as `pt` and `pts`, often occur in the same program. A good hash function would minimize the chance that such variants hash to the same slot.

A good approach is to derive the hash value in a way that is expected 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 frequently gives good results, assuming that the prime number is chosen to be unrelated to any patterns in the distribution of keys.

Finally, we note that some applications of hash functions might require stronger properties than are provided by simple uniform hashing. For example, we might want keys that are "close" in some sense to yield hash values that are far apart. (This property is especially desirable when we are using linear probing, defined in [Section 11.4](#).) Universal hashing, described in [Section 11.3.3](#), often provides the desired properties.

Interpreting keys as natural numbers

Most hash functions assume that the universe of keys is the set $N = \{0, 1, 2, \dots\}$ of natural numbers. Thus, if the keys are not natural numbers, a way is found to interpret them as natural numbers. For example, a character string can be interpreted as an integer expressed in suitable radix notation. Thus, the identifier `pt` might be interpreted as the pair of decimal integers (112, 116), since `p` = 112 and `t` = 116 in the ASCII character set; then, expressed as a radix-128 integer, `pt` becomes $(112 \cdot 128) + 116 = 14452$. It is usually straightforward in an application to devise some such method for interpreting each key as a (possibly large) natural number. In what follows, we assume that the keys are natural numbers.

11.3.1 The division method

In the *division method* for creating hash functions, we map 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.

When using the division method, we usually avoid certain values of m . For example, m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k . Unless it is known that all low-order p -bit patterns are equally likely, it is better to make the hash function depend on all the bits of the key. As [Exercise 11.3-3](#) asks you to show, choosing $m = 2^p - 1$ when k is a character string interpreted in radix 2^p may be a poor choice, because permuting the characters of k does not change its hash value.

A prime not too close to an exact power of 2 is often a good choice for m . For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2000$ character strings, where a character has 8 bits. We don't mind examining an average

of 3 elements in an unsuccessful search, so we allocate a hash table of size $m = 701$. The number 701 is chosen because it is a prime near $2000/3$ but not near any power of 2. Treating each key k as an integer, our hash function would be

$$h(k) = k \bmod 701.$$

11.3.2 The multiplication method

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

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

where " $kA \bmod 1$ " means the fractional part of kA , that is, $kA - \lfloor kA \rfloor$.

An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2 ($m = 2^p$ for some integer p) since we can then easily implement the function on most computers as follows. Suppose that the word size of the machine is w bits and that k fits into a single word. We restrict A to be a fraction of the form $s/2^w$, where s is an integer in the range $0 < s < 2^w$. Referring to [Figure 11.4](#), we first multiply k by the w -bit integer $s = A \cdot 2^w$. The result is a $2w$ -bit value $r_12^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word of the product. The desired p -bit hash value consists of the p most significant bits of r_0 .

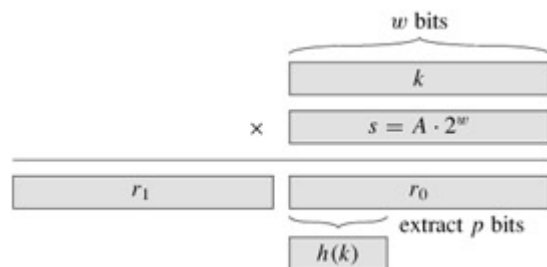


Figure 11.4: The multiplication method of hashing. The w -bit representation of the key k is multiplied by the w -bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w -bit half of the product form the desired hash value $h(k)$.

Although this method works with any value of the constant A , it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. [Knuth \[185\]](#) suggests that

$$(11.2) \quad A \approx (\sqrt{5} - 1)/2 = 0.6180339887 \dots$$

is likely to work reasonably well.

As an example, suppose we have $k = 123456$, $p = 14$, $m = 2^{14} = 16384$, and $w = 32$. Adapting Knuth's suggestion, we choose A to be the fraction of the form $s/2^{32}$ that is closest to $(\sqrt{5} - 1)/2$, so that $A = 2654435769/2^{32}$. Then $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, and so $r_1 = 76300$ and $r_0 = 17612864$. The 14 most significant bits of r_0 yield the value $h(k) = 67$.

11.3.3 □ Universal hashing

If a malicious adversary chooses the keys to be hashed by some fixed hash function, then he can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Any fixed 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, called **universal hashing**, can yield provably good performance on average, no matter what keys are chosen by the adversary.

The main idea behind universal hashing is to select the hash function at random from a carefully designed class of functions at the beginning of execution. As in the case of quicksort, randomization guarantees that no single input will always evoke worst-case behavior. Because of the randomization, the algorithm can behave differently on each execution, even for the same input, guaranteeing good average-case performance for any input. Returning to the example of a compiler's symbol table, we find that the programmer's choice of identifiers cannot now cause consistently poor hashing performance. Poor performance occurs only when the compiler chooses a random hash function that causes the set of identifiers to hash poorly, but the probability of this situation occurring is small and is the same for any set of identifiers of the same size.

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

The following theorem shows that a universal class of hash functions gives good average-case behavior. Recall that n_i denotes the length of list $T[i]$.

Theorem 11.3

Suppose that a hash function h is chosen from a universal collection of hash functions and is used to hash n keys into a table T of size m , using chaining to resolve collisions. If key k is not in the table, then the expected length $E[n_{h(k)}]$ of the list that key k hashes to is at most α . If key k is in the table, then the expected length $E[n_{h(k)}]$ of the list containing key k is at most $1 + \alpha$.

Proof We note that the expectations here are over the choice of the hash function, and do not depend on any assumptions about the distribution of the keys. For each pair k and l of distinct keys, define the indicator random variable $X_{kl} = I\{h(k) = h(l)\}$. Since by definition, a single pair of keys collides with probability at most $1/m$, we have $\Pr\{h(k) = h(l)\} \leq 1/m$, and so [Lemma 5.1](#) implies that $E[X_{kl}] \leq 1/m$.

Next we define, for each key k , the random variable Y_k that equals the number of keys other than k that hash to the same slot as k , so that

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}.$$

Thus we have

$$\begin{aligned} E[Y_k] &= E\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\ &= \sum_{\substack{l \in T \\ l \neq k}} E[X_{kl}] \quad (\text{by linearity of expectation}) \\ &\leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}. \end{aligned}$$

The remainder of the proof depends on whether key k is in table T .

- If $k \notin T$, then $n_{h(k)} = Y_k$ and $|\{l : l \in T \text{ and } l \neq k\}| = n$. Thus $E[n_{h(k)}] = E[Y_k] \leq n/m = \alpha$.
- If $k \in T$, then because key k appears in list $T[h(k)]$ and the count Y_k does not include key k , we have $n_{h(k)} = Y_k + 1$ and $|\{l : l \in T \text{ and } l \neq k\}| = n - 1$. Thus $E[n_{h(k)}] = E[Y_k] + 1 \leq (n - 1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$.

The following corollary says universal hashing provides the desired payoff: it is now impossible for an adversary to pick a sequence of operations that forces the worst-case running time. By cleverly randomizing the choice of hash function at run time, we guarantee that every sequence of operations can be handled with good expected running time.

Corollary 11.4

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

Proof Since the number of insertions is $O(m)$, we have $n = O(m)$ and so $\alpha = O(1)$. The INSERT and DELETE operations take constant time and, by [Theorem 11.3](#), the expected time for each SEARCH operation is $O(1)$. By linearity of expectation, therefore, the expected time for the entire sequence of operations is $O(n)$.

Designing a universal class of hash functions

It is quite easy to design a universal class of hash functions, as a little number theory will help us prove. You may wish to consult [Chapter 31](#) first if you are unfamiliar with number theory.

We begin by choosing a prime number p large enough so that every possible key k is in the range 0 to $p - 1$, inclusive. Let \mathbf{Z}_p denote the set $\{0, 1, \dots, p - 1\}$, and let \mathbf{Z}_p^* denote the set $\{1, 2, \dots, p - 1\}$. Since p is prime, we can solve equations modulo p with the methods given in [Chapter 31](#). Because we assume that the size of the universe of keys is greater than the number of slots in the hash table, we have $p > m$.

We now define the hash function $h_{a,b}$ for any $a \in \mathbf{Z}_p^*$ and any $b \in \mathbf{Z}_p$ using a linear transformation followed by reductions modulo p and then modulo m :

$$(11.3) \quad h_{a,b}(k) = ((ak + b) \bmod p) \bmod m.$$

For example, with $p = 17$ and $m = 6$, we have $h_{3,4}(8) = 5$. The family of all such hash functions is

$$(11.4) \quad \mathcal{H}_{p,m} = \{h_{a,b} : a \in \mathbf{Z}_p^* \text{ and } b \in \mathbf{Z}_p\}.$$

Each hash function $h_{a,b}$ maps \mathbf{Z}_p to \mathbf{Z}_m . This class of hash functions has the nice property that the size m of the output range is arbitrary—not necessarily prime—a feature which we shall use in [Section 11.5](#). Since there are $p - 1$ choices for a and there are p choices for b , there are $p(p - 1)$ hash functions in $\mathcal{H}_{p,m}$.

Theorem 11.5

The class $\mathcal{H}_{p,m}$ of hash functions defined by equations [\(11.3\)](#) and [\(11.4\)](#) is universal.

Proof Consider two distinct keys k and l from \mathbf{Z}_p , so that $k \neq l$. For a given hash function $h_{a,b}$ we let

$$\begin{aligned} r &= (ak + b) \bmod p, \\ s &= (al + b) \bmod p. \end{aligned}$$

We first note that $r \neq s$. Why? Observe that

$$r - s \equiv a(k - l) \pmod{p}.$$

It follows that $r \neq s$ because p is prime and both a and $(k - l)$ are nonzero modulo p , and so their product must also be nonzero modulo p by [Theorem 31.6](#). Therefore, during the computation of any $h_{a,b}$ in $\mathcal{H}_{p,m}$, distinct inputs k and l map to distinct values r and s modulo p ; 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, s) with $r \neq s$, since we can solve for a and b given r and s :

$$\begin{aligned} a &= ((r - s)((k - l)^{-1} \bmod p)) \bmod p, \\ b &= (r - ak) \bmod p, \end{aligned}$$

where $((k - l)^{-1} \bmod p)$ denotes the unique multiplicative inverse, modulo p , of $k - l$. Since there are only $p(p - 1)$ possible pairs (r, s) with $r \neq s$, there is a one-to-one correspondence

between pairs (a, b) with $a \neq 0$ and pairs (r, s) with $r \neq s$. Thus, for any given pair of inputs k and l , if we pick (a, b) uniformly at random from $\mathbf{Z}_p^* \times \mathbf{Z}_p$, the resulting pair (r, s) is equally likely to be any pair of distinct values modulo p .

It then follows that the probability that distinct keys k and l collide is equal to the probability that $r \equiv s \pmod{m}$ when r and s are randomly chosen as distinct values modulo p . For a given value of r , of the $p - 1$ possible remaining values for s , the number of values s such that $s \neq r$ and $s \equiv r \pmod{m}$ is at most

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p + m - 1)/m) - 1 \text{ (by inequality (3.7))} \\ &= (p - 1)/m. \end{aligned}$$

The probability that s collides with r when reduced modulo m is at most $((p - 1)/m)/(p - 1) = 1/m$.

Therefore, for any pair of distinct values $k, l \in \mathbf{Z}_p$,

$$\Pr\{h_{a,b}(k) = h_{a,b}(l)\} \leq 1/m,$$

so that $\mathcal{H}_{p,m}$ is indeed universal.

Exercises 11.3-1

Suppose we 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 we take advantage of the hash values when searching the list for an element with a given key?

Exercises 11.3-2

Suppose that a string of r characters is hashed into m slots by treating it as a radix-128 number and then using the division method. The number m is easily represented as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can we 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?

Exercises 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 derived from 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.

Exercises 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.

Exercises 11.3-5: ★

Define a family \mathcal{H} of hash functions from a finite set U to a finite set B to be ϵ -**universal** if for all pairs of distinct elements k and l in U ,

$$\Pr \{h(k) = h(l)\} \leq \epsilon,$$

where the probability is taken over the drawing of hash function h at random from the family \mathcal{H} . Show that an ϵ -universal family of hash functions must have

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

Exercises 11.3-6: ★

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

$$h_b(\langle a_0, a_1, \dots, a_{n-1} \rangle) = \sum_{j=0}^{n-1} a_j b^j$$

and let $\mathcal{H} = \{h_b : b \in \mathbb{Z}_p\}$. Argue that \mathcal{H} is $((n-1)/p)$ -universal according to the definition of ϵ -universal in [Exercise 11.3-5](#). (Hint: See [Exercise 31.4-4](#).)

11.4 Open addressing

In **open addressing**, all elements are stored in the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until the desired element is found or it is clear that the element is not in the table. There are no lists and no elements stored outside the table, as there

are in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made; the load factor α can never exceed 1.

Of course, we could store the linked lists for chaining 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, we *compute* the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

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

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}.$$

With open addressing, we require that for every key k , the **probe sequence**

$$\square h(k, 0), h(k, 1), \dots, h(k, m - 1) \square$$

be a permutation of $\square 0, 1, \dots, m - 1 \square$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. In the following pseudocode, we assume 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).

```

HASH-INSERT( $T, k$ )
1   $i \leftarrow 0$ 
2  repeat  $j \leftarrow h(k, i)$ 
3      if  $T[j] = \text{NIL}$ 
4          then  $T[j] \leftarrow k$ 
5              return  $j$ 
6      else  $i \leftarrow i + 1$ 
7  until  $i = m$ 
8  error "hash table overflow"

```

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. (This argument assumes that keys are not deleted from the hash table.) The procedure HASH-SEARCH takes as input a hash table T and a key k , returning j if slot j is found to contain key k , or NIL if key k is not present in table T .

```

HASH-SEARCH( $T, k$ )
1   $i \leftarrow 0$ 
2  repeat  $j \leftarrow h(k, i)$ 
3      if  $T[j] = k$ 
4          then return  $j$ 
5       $i \leftarrow i + 1$ 
6  until  $T[j] = \text{NIL}$  or  $i = m$ 

```


7 **return** NIL

Deletion from an open-address hash table is difficult. When we delete a key from slot i , we cannot simply mark that slot as empty by storing NIL in it. Doing so might make it impossible to retrieve any key k during whose insertion we had probed slot i and found it occupied. One solution is to mark the slot by storing in it the special value DELETED instead of NIL. We would then modify the procedure HASH-INSERT to treat such a slot as if it were empty so that a new key can be inserted. No modification of HASH-SEARCH is needed, since it will pass over DELETED values while searching. When we use the special value DELETED, however, search times are no longer dependent on the load factor α , and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

In our analysis, we make the assumption of **uniform hashing**: we assume that each key is equally likely to have any of the $m!$ permutations of $\{0, 1, \dots, m-1\}$ as its probe sequence. Uniform hashing generalizes the notion of simple uniform hashing defined earlier to the situation in which the hash function produces not just a single number, but a whole probe sequence. True uniform hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used.

Three techniques are commonly used to compute the probe sequences required for open addressing: linear probing, quadratic probing, and double hashing. These techniques all guarantee that $\{h(k, 0), h(k, 1), \dots, h(k, m-1)\}$ is a permutation of $\{0, 1, \dots, m-1\}$ for each key k . None of these techniques fulfills the assumption of uniform hashing, however, since none of them is capable of generating more than m^2 different probe sequences (instead of the $m!$ that uniform hashing requires). Double hashing has the greatest number of probe sequences and, as one might expect, seems to give the best results.

Linear probing

Given an ordinary hash function $h' : U \rightarrow \{0, 1, \dots, m-1\}$, which we refer to as an **auxiliary hash function**, the method of **linear probing** uses the hash function

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

for $i = 0, 1, \dots, m-1$. Given key k , the first slot probed is $T[h'(k)]$, i.e., the slot given by the auxiliary hash function. We next probe slot $T[h'(k) + 1]$, and so on up to slot $T[m-1]$. Then we wrap around to slots $T[0]$, $T[1]$, ..., until we finally probe slot $T[h'(k) - 1]$. Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.

Linear probing is easy to implement, but it suffers from a problem known as **primary clustering**. Long runs of occupied slots build up, increasing the average search time. Clusters arise since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$. Long runs of occupied slots tend to get longer, and the average search time increases.

Quadratic probing

Quadratic probing uses a hash function of the form

$$(11.5) \quad h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

where h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \dots, m - 1$. The initial position probed is $T[h'(k)]$; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i . This method works much better than linear probing, but to make full use of the hash table, the values of c_1 , c_2 , and m are constrained. [Problem 11-3](#) shows one way to select these parameters. Also, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$. This property leads to a milder form of clustering, called **secondary clustering**. As in linear probing, the initial probe determines the entire sequence, so only m distinct probe sequences are used.

Double hashing

Double hashing is 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 h_1 and h_2 are auxiliary hash functions. The initial probe is to position $T[h_1(k)]$; successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m . Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k , since the initial probe position, the offset, or both, may vary. [Figure 11.5](#) gives an example of insertion by double hashing.

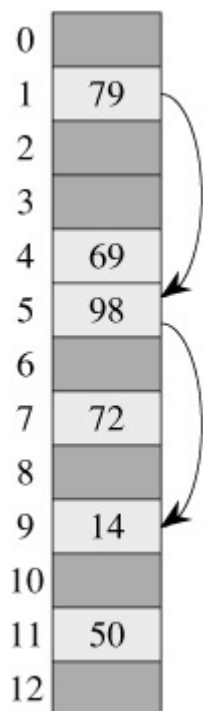


Figure 11.5: Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched. (See [Exercise 11.4-3](#).) A convenient way to ensure this condition is to let m be a 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, we could choose m prime and let

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

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

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

Double hashing improves over linear or quadratic probing in that $\Theta(m^2)$ probe sequences are used, rather than $\Theta(m)$, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence. As a result, the performance of double hashing appears to be very close to the performance of the "ideal" scheme of uniform hashing.

Analysis of open-address hashing

Our analysis of open addressing, like our analysis of chaining, is expressed in terms of the load factor $\alpha = n/m$ of the hash table, as n and m go to infinity. Of course, with open addressing, we have at most one element per slot, and thus $n \leq m$, which implies $\alpha \leq 1$.

We assume that uniform hashing is used. In this idealized scheme, the probe sequence $\square h(k, 0), h(k, 1), \dots, h(k, m - 1) \square$ used to insert or search for each key k is equally likely to be any permutation of $\square 0, 1, \dots, m - 1 \square$. Of course, a given key has a unique fixed probe sequence associated with it; what is meant 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 uniform hashing, beginning with an analysis of the number of probes made in an unsuccessful search.

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 uniform hashing.

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 us define the random variable X to be the number of probes made in an unsuccessful search, and let us also define the event A_i , for $i = 1, 2, \dots$, to be the event that there is an i th probe and it is to an occupied slot. Then the event $\{X \geq i\}$ is the intersection of events $A_1 \cap A_2 \cap \dots \cap A_{i-1}$. We will bound $\Pr\{X \geq i\}$ by bounding $\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\}$. By [Exercise C.2-6](#),

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

$$\Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \dots \cap A_{i-2}\}.$$

Since there are n elements and m slots, $\Pr \{A_1\} = n/m$. For $j > 1$, the probability that there is a j th probe and it is to an occupied slot, given that the first $j - 1$ probes were to occupied slots, is $(n - j + 1)/(m - j + 1)$. This probability follows because we would be finding one of the remaining $(n - (j - 1))$ elements in one of the $(m - (j - 1))$ unexamined slots, and by the assumption of uniform hashing, the probability is the ratio of these quantities. Observing that $n < m$ implies that $(n - j)/(m - j) \leq n/m$ for all j such that $0 \leq j < m$, we have for all i such that $1 \leq i \leq m$,

$$\begin{aligned} \Pr \{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1}. \end{aligned}$$

Now we use equation (C.24) to bound the expected number of probes:

$$\begin{aligned} E[X] &= \sum_{i=1}^{\infty} \Pr \{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^i \\ &= \frac{1}{1-\alpha}. \end{aligned}$$

The above bound of $1 + \alpha + \alpha^2 + \alpha^3 + \cdots$ has an intuitive interpretation. One probe is always made. With probability approximately α , the first probe finds an occupied slot so that a second probe is necessary. With probability approximately α^2 , the first two slots are occupied so that a third probe is necessary, and so on.

If α is a constant, [Theorem 11.6](#) predicts that an unsuccessful search runs in $O(1)$ time. For example, if the hash table is half full, the average number of probes in an unsuccessful search is at most $1/(1 - .5) = 2$. If it is 90 percent full, the average number of probes is at most $1/(1 - .9) = 10$.

[Theorem 11.6](#) gives us the performance of the HASH-INSERT procedure almost immediately.

Corollary 11.7

Inserting an element into an open-address hash table with load factor α requires at most $1/(1 - \alpha)$ probes on average, assuming uniform hashing.

Proof An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placement of the key in the first empty slot found. Thus, the expected number of probes is at most $1/(1 - \alpha)$.

Computing the expected number of probes for a successful search requires a little more work.

Theorem 11.8

Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha},$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Proof A search for a key k follows the same probe sequence as was followed when the element with key k was inserted. By [Corollary 11.7](#), if k was the $(i + 1)$ st key inserted into the hash table, the expected number of probes made in a search for k is at most $1/(1 - i/m) = m/(m - i)$. Averaging over all n keys in the hash table gives us the average number of probes in a successful search:

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha} (H_m - H_{m-n}), \end{aligned}$$

where $H_i = \sum_{j=1}^i 1/j$ is the i th harmonic number (as defined in equation [\(A.7\)](#)). Using the technique of bounding a summation by an integral, as described in [Section A.2](#), we obtain

$$\begin{aligned} \frac{1}{\alpha} (H_m - H_{m-n}) &= \frac{1}{\alpha} \sum_{k=m-n+1}^m 1/k \\ &\leq \frac{1}{\alpha} \int_{m-n}^m (1/x) dx \quad (\text{by inequality (A.12)}) \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} \\ &= \frac{1}{\alpha} \ln \frac{1}{1 - \alpha} \end{aligned}$$

for a bound on the expected number of probes in a successful search.

If the hash table is half full, the expected number of probes in a successful search is less than 1.387. If the hash table is 90 percent full, the expected number of probes is less than 2.559.

Exercises 11.4-1

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the primary hash function $h'(k) = k \bmod m$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_2(k) = 1 + (k \bmod (m - 1))$.

Exercises 11.4-2

Write pseudocode for HASH-DELETE as outlined in the text, and modify HASH-INSERT to handle the special value DELETED.

Exercises 11.4-3: ★

Suppose that we use double hashing to resolve collisions; that is, we use the hash function $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Show that if m and $h_2(k)$ have greatest common divisor $d \geq 1$ for some key k , then an unsuccessful search for key k examines $(1/d)$ th of the hash table before returning to slot $h_1(k)$. Thus, when $d = 1$, so that m and $h_2(k)$ are relatively prime, the search may examine the entire hash table. (*Hint:* See [Chapter 31](#).)

Exercises 11.4-4

Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is $3/4$ and when it is $7/8$.

Exercises 11.4-5: ★

Consider an open-address hash table with a load factor α . Find the nonzero value α for which the expected number of probes in an unsuccessful search equals twice the expected number of probes in a successful search. Use the upper bounds given by [Theorems 11.6](#) and [11.8](#) for these expected numbers of probes.

11.5 □ Perfect hashing

Although hashing is most often used for its excellent expected performance, hashing can be used to obtain excellent *worst-case* performance when the set of keys is *static*: once the keys

are stored in the table, the set of keys never changes. Some applications naturally have static sets of keys: consider the set of reserved words in a programming language, or the set of file names on a CD-ROM. We call a hashing technique **perfect hashing** if the worst-case number of memory accesses required to perform a search is $O(1)$.

The basic idea to create a perfect hashing scheme is simple. We use a two-level hashing scheme with universal hashing at each level. [Figure 11.6](#) illustrates the approach.

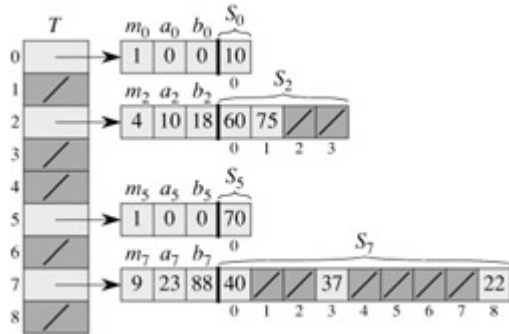


Figure 11.6: Using perfect hashing to store the set $K = \{10, 22, 37, 40, 60, 70, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T . A secondary hash table S_j stores all keys hashing to slot j . The size of hash table S_j is m_j , and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 1$, key 75 is stored in slot 1 of secondary hash table S_2 . There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.

The first level is essentially the same as for hashing with chaining: the n keys are hashed into m slots using a hash function h carefully selected from a family of universal hash functions.

Instead of making a list of the keys hashing to slot j , however, we use a small **secondary hash table** S_j with an associated hash function h_j . By choosing the hash functions h_j carefully, we can guarantee that there are no collisions at the secondary level.

In order to guarantee that there are no collisions at the secondary level, however, we will need to let the size m_j of hash table S_j be the square of the number n_j of keys hashing to slot j . While having such a quadratic dependence of m_j on n_j may seem likely to cause the overall storage requirements to be excessive, we shall show that by choosing the first level hash function well, the expected total amount of space used is still $O(n)$.

We use hash functions chosen from the universal classes of hash functions of [Section 11.3.3](#).

The first-level hash function is chosen from the class $\mathcal{H}_{p,m}$, where as in [Section 11.3.3](#), p is a prime number greater than any key value. Those keys hashing to slot j are re-hashed into a secondary hash table S_j of size m_j using a hash function h_j chosen from the class \mathcal{H}_{p,m_j} .^[1]

We shall proceed in two steps. First, we shall determine how to ensure that the secondary tables have no collisions. Second, we shall show that the expected amount of memory used overall—for the primary hash table and all the secondary hash tables—is $O(n)$.

Theorem 11.9

If we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions, then the probability of there being any collisions is less than $1/2$.

Proof There are $\binom{n}{2}$ pairs of keys that may collide; each pair collides with probability $1/m$ if h is chosen at random from a universal family \mathcal{H} of hash functions. Let X be a random variable that counts the number of collisions. When $m = n^2$, the expected number of collisions is

$$\begin{aligned} E[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2. \end{aligned}$$

(Note that this analysis is similar to the analysis of the birthday paradox in [Section 5.4.1](#).) Applying Markov's inequality ([C.29](#)), $\Pr\{X \geq t\} \leq E[X]/t$, with $t = 1$ completes the proof.

In the situation described in [Theorem 11.9](#), where $m = n^2$, it follows that a hash function h chosen at random from \mathcal{H} is more likely than not to have *no* collisions. Given the set K of n keys to be hashed (remember that K is static), it is thus easy to find a collision-free hash function h with a few random trials.

When n is large, however, a hash table of size $m = n^2$ is excessive. Therefore, we adopt the two-level hashing approach, and we use the approach of [Theorem 11.9](#) only to hash the entries within each slot. An outer, or first-level, hash function h is used to hash the keys into $m = n$ slots. Then, if n_j keys hash to slot j , a secondary hash table S_j of size $m_j = n_j^2$ is used to provide collision-free constant time lookup.

We now turn to the issue of ensuring that the overall memory used is $O(n)$. Since the size m_j of the j th secondary hash table grows quadratically with the number n_j of keys stored, there is a risk that the overall amount of storage could be excessive.

If the first-level table size is $m = n$, then the amount of memory used is $O(n)$ for the primary hash table, for the storage of the sizes m_j of the secondary hash tables, and for the storage of the parameters a_j and b_j defining the secondary hash functions h_j drawn from the class \mathcal{H}_{p, m_j} of [Section 11.3.3](#) (except when $n_j = 1$ and we use $a = b = 0$). The following theorem and a corollary provide a bound on the expected combined sizes of all the secondary hash tables. A second corollary bounds the probability that the combined size of all the secondary hash tables is superlinear.

Theorem 11.10

If we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions, then

$$\mathbb{E} \left[\sum_{j=0}^{m-1} n_j^2 \right] < 2n ,$$

where n_j is the number of keys hashing to slot j .

Proof We start with the following identity, which holds for any nonnegative integer a :

$$(11.6) \quad a^2 = a + 2 \binom{a}{2} .$$

We have

$$\begin{aligned} \mathbb{E} \left[\sum_{j=0}^{m-1} n_j^2 \right] &= \mathbb{E} \left[\sum_{j=0}^{m-1} \left(n_j + 2 \binom{n_j}{2} \right) \right] && \text{(by equation (11.6))} \\ &= \mathbb{E} \left[\sum_{j=0}^{m-1} n_j \right] + 2 \mathbb{E} \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] && \text{(by linearity of expectation)} \\ &= \mathbb{E}[n] + 2 \mathbb{E} \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] && \text{(by equation (11.1))} \\ &= n + 2 \mathbb{E} \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] && \text{(since } n \text{ is not a random variable) .} \end{aligned}$$

To evaluate the summation $\sum_{j=0}^{m-1} \binom{n_j}{2}$, we observe that it is just the total number of collisions. By the properties of universal hashing, the expected value of this summation is at most

$$\binom{n}{2} \frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2} ,$$

since $m = n$. Thus,

$$\begin{aligned} \mathbb{E} \left[\sum_{j=0}^{m-1} n_j^2 \right] &\leq n + 2 \frac{n-1}{2} \\ &= 2n - 1 \\ &< 2n . \end{aligned}$$

Corollary 11.11

If we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions and we set the size of each secondary hash table to $m_j = n_j^2$ for $j = 0, 1, \dots, m-1$, then the expected amount of storage required for all secondary hash tables in a perfect hashing scheme is less than $2n$.

Proof Since $m_j = n_j^2$ for $j = 0, 1, \dots, m-1$, [Theorem 11.10](#) gives

$$(11.7) \quad E \left[\sum_{j=0}^{m-1} m_j \right] = E \left[\sum_{j=0}^{m-1} n_j^2 \right] < 2n ,$$

which completes the proof.

Corollary 11.12

If we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions and we set the size of each secondary hash table to $m_j = n_j^2$ for $j = 0, 1, \dots, m - 1$, then the probability that the total storage used for secondary hash tables exceeds $4n$ is less than $1/2$.

Proof Again we apply Markov's inequality ([C.29](#)), $\Pr \{X \geq t\} \leq E[X] / t$, this time to inequality ([11.7](#)), with $X = \sum_{j=0}^{m-1} m_j$ and $t = 4n$:

$$\begin{aligned} \Pr \left\{ \sum_{j=0}^{m-1} m_j \geq 4n \right\} &\leq \frac{E \left[\sum_{j=0}^{m-1} m_j \right]}{4n} \\ &< \frac{2n}{4n} \\ &= 1/2 . \end{aligned}$$

From [Corollary 11.12](#), we see that testing a few randomly chosen hash functions from the universal family will quickly yield one that uses a reasonable amount of storage.

Exercises 11.5-1: ★

Suppose that we insert n keys into a hash table of size m using open addressing and uniform hashing. Let $p(n, m)$ be the probability that no collisions occur. Show that $p(n, m) \leq e^{-n(n-1)/2m}$. (Hint: See [equation \(3.11\)](#).) Argue that when n exceeds \sqrt{m} , the probability of avoiding collisions goes rapidly to zero.

Problems 11-1: Longest-probe bound for hashing

A hash table of size m is used to store n items, with $n \leq m/2$. Open addressing is used for collision resolution.

- Assuming uniform hashing, show that for $i = 1, 2, \dots, n$, the probability that the i th insertion requires strictly more than k probes is at most 2^{-k} .
- Show that for $i = 1, 2, \dots, n$, the probability that the i th insertion requires more than $2 \lg n$ probes is at most $1/n^2$.

Let the random variable X_i denote the number of probes required by the i th insertion. You have shown in part (b) that $\Pr \{X_i > 2 \lg n\} \leq 1/n^2$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

- c. Show that $\Pr\{X > 2 \lg n\} \leq 1/n$.
- d. Show that the expected length $E[X]$ of the longest probe sequence is $O(\lg n)$.

Problems 11-2: Slot-size bound for chaining

Suppose that we have a hash table with n slots, with collisions resolved by chaining, and suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission is to prove an $O(\lg n / \lg \lg n)$ upper bound on $E[M]$, the expected value of M .

- a. Argue that the probability Q_k that exactly k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

- b. Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \leq nQ_k$.
- c. Use Stirling's approximation, [equation \(3.17\)](#), to show that $Q_k < e^k/k^k$.
- d. Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^2$ for $k_0 = c \lg n / \lg \lg n$. Conclude that $P_k < 1/n^2$ for $k \geq k_0 = c \lg n / \lg \lg n$.
- e. Argue that

$$E[M] \leq \Pr \left\{ M > \frac{c \lg n}{\lg \lg n} \right\} \cdot n + \Pr \left\{ M \leq \frac{c \lg n}{\lg \lg n} \right\} \cdot \frac{c \lg n}{\lg \lg n}.$$

Conclude that $E[M] = O(\lg n / \lg \lg n)$.

Problems 11-3: Quadratic probing

Suppose that we are given a key k to search for in a hash table with positions $0, 1, \dots, m-1$, and suppose that we have a hash function h mapping the key space into the set $\{0, 1, \dots, m-1\}$. The search scheme is as follows.

1. Compute the value $i \leftarrow h(k)$, and set $j \leftarrow 0$.
2. Probe in position i for the desired key k . If you find it, or if this position is empty, terminate the search.
3. Set $j \leftarrow (j + 1) \bmod m$ and $i \leftarrow (i + j) \bmod m$, and return to step 2.

Assume that m is a power of 2.

- Show that this scheme is an instance of the general "quadratic probing" scheme by exhibiting the appropriate constants c_1 and c_2 for [equation \(11.5\)](#).
- Prove that this algorithm examines every table position in the worst case.

Problems 11-4: k-universal hashing and authentication

Let $\mathcal{H} = \{h\}$ be a class of hash functions in which each h maps the universe U of keys to $\{0, 1, \dots, m-1\}$. We say that \mathcal{H} is **k-universal** if, for every fixed sequence of k distinct keys $\langle x^{(1)}, x^{(2)}, \dots, x^{(k)} \rangle$ and for any h chosen at random from \mathcal{H} , the sequence $\langle h(x^{(1)}), h(x^{(2)}), \dots, h(x^{(k)}) \rangle$ is equally likely to be any of the m^k sequences of length k with elements drawn from $\{0, 1, \dots, m-1\}$.

- Show that if \mathcal{H} is 2-universal, then it is universal.
- Let U be the set of n -tuples of values drawn from \mathbf{Z}_p , and let $B = \mathbf{Z}_p$, where p is prime. For any n -tuple $a = \langle a_0, a_1, \dots, a_{n-1} \rangle$ of values from \mathbf{Z}_p and for any $b \in \mathbf{Z}_p$, define the hash function $h_{a,b} : U \rightarrow B$ on an input n -tuple $x = \langle x_0, x_1, \dots, x_{n-1} \rangle$ from U as

$$h_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

and let $\mathcal{H} = \{h_{a,b}\}$. Argue that \mathcal{H} is 2-universal.

- Suppose that Alice and Bob agree secretly on a hash function $h_{a,b}$ from a 2-universal family \mathcal{H} of hash functions. Later, Alice sends a message m to Bob over the Internet, where $m \in U$. She authenticates this message to Bob by also sending an authentication tag $t = h_{a,b}(m)$, and Bob checks that the pair (m, t) he receives satisfies $t = h_{a,b}(m)$. Suppose that an adversary intercepts (m, t) en route and tries to fool Bob by replacing the pair with a different pair (m', t') . Argue that the probability that the adversary succeeds in fooling Bob into accepting (m', t') is at most $1/p$, no matter how much computing power the adversary has.

^[1]When $n_j = m_j = 1$, we don't really need a hash function for slot j ; when we Choose a hash function $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m_j$ for such a slot, we just use $a = b = 0$.

Chapter notes

[Knuth \[185\]](#) and [Gonnet \[126\]](#) are excellent references for the analysis of hashing algorithms. Knuth credits H. P. Luhn (1953) for inventing hash tables, along with the chaining method for resolving collisions. At about the same time, G. M. Amdahl originated the idea of open addressing.

Carter and Wegman introduced the notion of universal classes of hash functions in 1979 [52].

Fredman, Komlós, and Szemerédi [96] developed the perfect hashing scheme for static sets presented in Section 11.5. An extension of their method to dynamic sets, handling insertions and deletions in amortized expected time $O(1)$, has been given by Dietzfelbinger et al. [73].

Chapter 12: Binary Search Trees

Overview

Search trees are data structures that support many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, a search tree can be used both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time. We shall see in Section 12.4 that the expected height of a randomly built binary search tree is $O(\lg n)$, so that basic dynamic-set operations on such a tree take $\Theta(\lg n)$ time on average.

In practice, we can't always guarantee that binary search trees are built randomly, but there are variations of binary search trees whose worst-case performance on basic operations can be guaranteed to be good. Chapter 13 presents one such variation, red-black trees, which have height $O(\lg n)$. Chapter 18 introduces B-trees, which are particularly good for maintaining databases on random-access, secondary (disk) storage.

After presenting the basic properties of binary search trees, the following sections show how to walk a binary search tree to print its values in sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor or successor of an element, and how to insert into or delete from a binary search tree. The basic mathematical properties of trees appear in Appendix B.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. Such a tree can be represented by a linked data structure in which each node is an object. In addition to a *key* field and satellite data, each node contains fields *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL.