Hash Tables

11.1 Direct Address Table

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 $U = \{0, 1, ..., m-1\}$, 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 $T[0..m-1]$, 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 $T[k] =$ NIL.

The dictionary operations are trivial to implement:

DIRECT-ADDRESS-SEARCH

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE

1 $T[x.key] =$ NIL

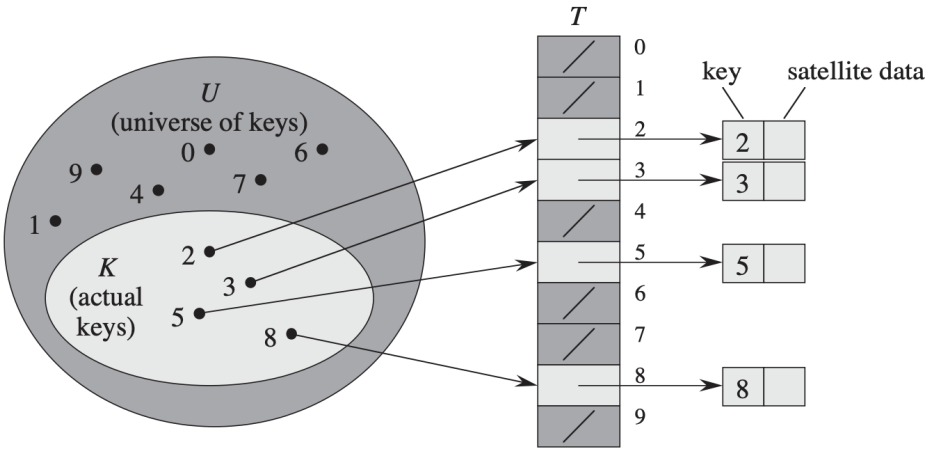

Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,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

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, we can store the object in the slot itself, thus saving space. We would use a special key within an object to indicate an empty slot. Moreover, it is often unnecessary to store the key 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 whether the slot is empty.

11.2 Hash Tables

The downside of 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 dictionary in much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, we can reduce the storage requirement to $\Theta(|K|)$ while we maintain 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*, 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*** $T[0\,..\,m-1]$:
$h : U \to \{0, 1, ..., m-1\}$,
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 $h(k)$; 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.

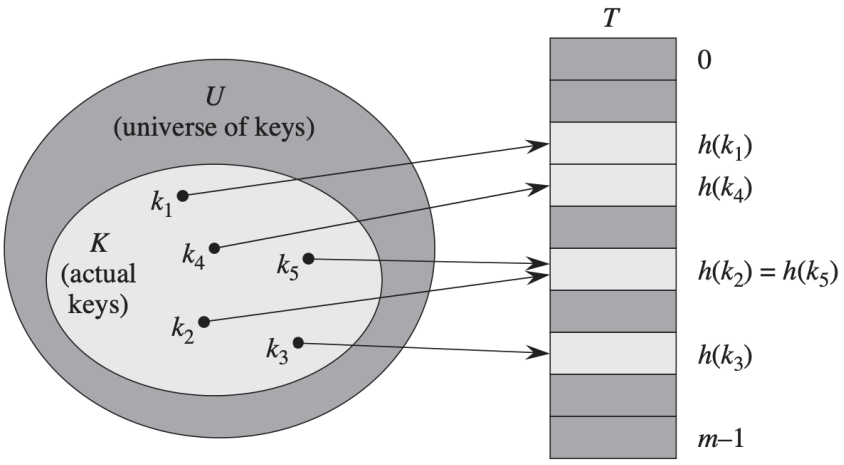

Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

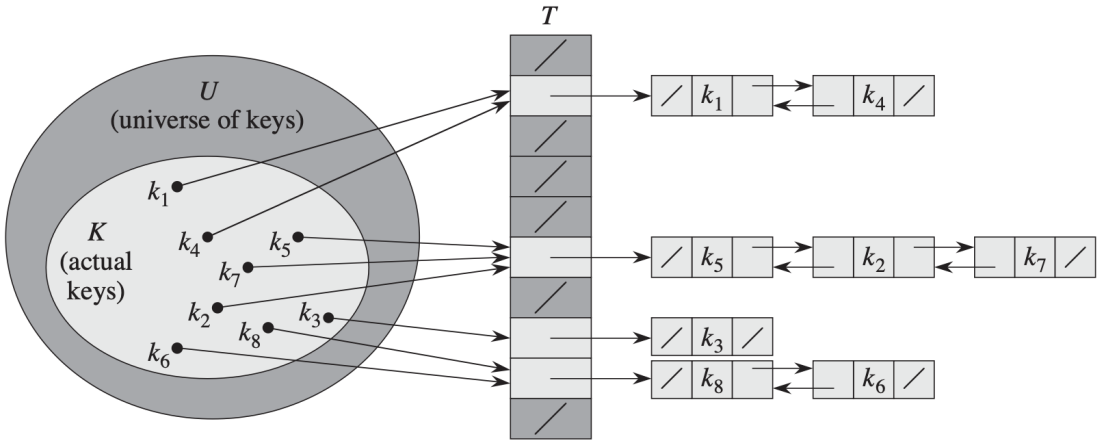

Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***. Fortunately, we have 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 $h(k)$). Because $|U| > m$, however, there must be at least two keys that have the same hash value; avoiding collisions altogether 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.

Chaining is a simple way to avoid collision.

11.2.1 Collision resolution by chaining

In ***chaining***, we place all the elements that hash to the same slot into the same linked list, as Figure 11.3 shows. 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.

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

CHAINED-HASH-INSERT

1 insert x at the head of list $T[h(x.key)]$

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

CHAINED-HASH-DELETE
1 delete x from the list $T[h(x.key)]$

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; if necessary, we can check this assumption (at additional cost) by searching for an element whose key is $x.key$ before we insert. For searching, the worst-case running time is proportional to the length of the list; we shall analyze this operation more closely below. We can delete an element in $O(1)$ time if the lists are doubly linked, as Figure 11.3 depicts. (Note that CHAINED-HASH-DELETE takes as input an element x and not its key k, so that we don't have to search for x first. If the hash table supports deletion, then its linked lists should be doubly linked so that we can delete an item quickly. If the lists were only singly linked, then to delete element x, we would first have to find x in the list $T[h(x.key)]$ so that we could update the *next* attribute of x's predecessor. With singly linked lists, both deletion and searching would have the same asymptotic running times.)

11.2.2 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 give 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, we do not use hash tables for their worst-case performance. (Perfect hashing does provide good worst-case performance when the set of keys is static, however.)

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. 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 assumption of ***simple uniform hashing***.

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

$$n = n_0 + n_1 + ... + n_{m-1} \tag{1}$$

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 $h(k)$, 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 the algorithm checks to see whether any have a key equal to 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 average-case 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 $h(k) = O(1)$ is $\Theta(1+\alpha)$.

The situation for a successful search is slightly different, since each list is not equally liked 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 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 average-case time $\Theta(1+\alpha)$, 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 one 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. 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 ith element inserted into the table, for $i = 1, 2, ..., n$, and let $k_i = x_i.key$. For keys k_i and k_j, we define the indicator random variable

$$X_{ij} = \begin{cases} 1 & \text{if the keys of } x_i \text{ and } x_j \text{ hash to the same table slot} \\ 0 & \text{otherwise} \end{cases}$$

Then if we want to count the number of keys that hash to the same list as element i *after* i is added to the list, that is just

$$\sum_{j=i+1}^{n} X_{ij}.$$

We are starting the sum at $i+1$ because we only want to count the elements that are inserted at i.

The search starts with a key, k_i. We first calculate $h[k_i]$, and then access $T[h(k_i)]$ in the hash table. In case of chaining, we may have multiple keys stored in the same location,

$T[h(k_i)]$ because of the chaining. That number of counts of elements with the same hash, $h(k_i)$, will be exactly $1 + \sum_{j=i+1}^{n} X_{ij}$.

Under the assumption of simple uniform hashing, we have $\Pr\{h(k_i) = h(k_j)\} = 1/m$, so $E[X_{ij}] = 1/m$. So the expected number of elements examined in a successful search is

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

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

Conclusion of this analysis:

- 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, we can support all dictionary operations in $O(1)$ time on average.

11.3 Hash functions

A good hash functions satisfies (approximately) the assumptions 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, we typically have no way to check this condition, since

we rarely know the probability distribution from which keys are drawn. Moreover, the keys might not be drawn independently. But we can frequently get good results by attempting to derive the hash value in a way that we expect to be independent of any patterns that might exist in the data.

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

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

satisfies the condition of simple uniform hashing.

11.3.1 The division method

Let $h(k) = k \bmod m$ for some value of m.

For instance, if $m = 12$, and $k = 100$, then $h(k) = 4$.

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, and generally not all low-order bit patterns are equally likely. (Explained here.)

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

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

where "$kA \mod 1$" 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 $r_1 2^w + r_0$, 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.

Here the value of m is not critical, but some values of A work better than others.

In other words, let $h(k) = \lfloor m(kA \bmod 1) \rfloor$, where $kA \mod 1$ = fractional part of kA. The optimal choice depends on the characteristics of the data being hashed.

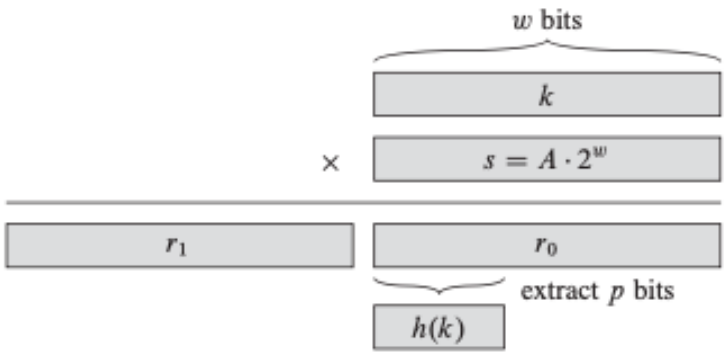

Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

Knuth suggests that $A = (\sqrt{5} - 1)/2$ is likely to work reasonably well.