1 HASH TABLES

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 1.1 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 1.2 presents the main ideas, and Section 1.3 describes how array indices can be computed from keys using hash functions. Hashing is an extremely effective and practical technique: the basic dictionary operations require only $O(1)$ time on the average.

1.1 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, . . . , m - 1}, where m is not too large. We shall assume that no two elements have the same key.

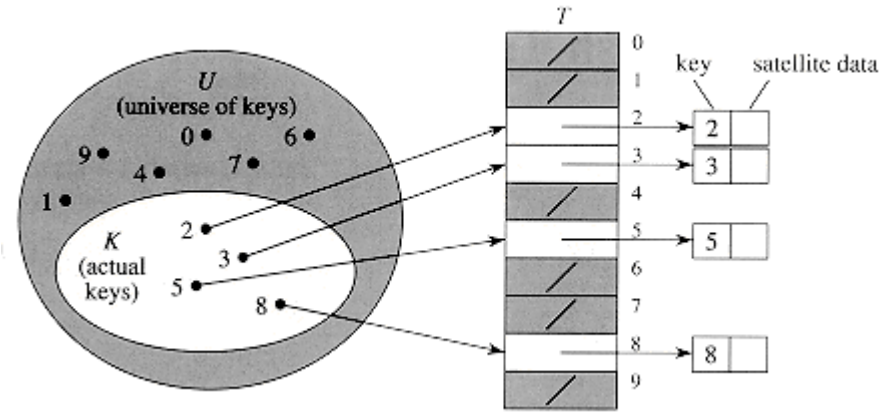

Figure 1.1 Implementing a dynamic set by a direct-address table T. Each key in the universe U = {0,1, . . . , 9} corresponds to an index in the table. The set K = {2, 3, 5, 8} of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

To represent the dynamic set, we use an array, or direct-address table, T [0 . . m - 1], in which each position, or slot, corresponds to a key in the universe U. Figure 1.1 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 -

DIRECT-ADDRESS-SEARCH(T,k)

return T[k]

DIRECT-ADDRESS-INSERT(T,x)

T[key[x]] ← x

DIRECT-ADDRESS-DELETE(T,x)

T[key[x]] ← 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.

1.2 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 $\Theta(|K|)$, even though 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.)

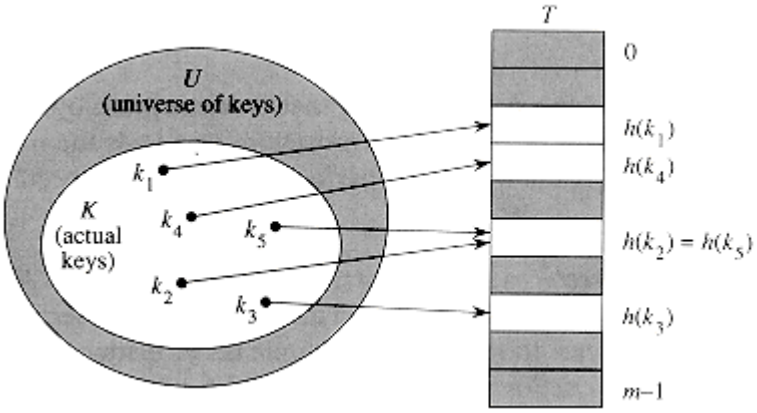

Figure 1.2 Using a hash function h to map keys to hash-table slots. Keys k2 and k5 map to the same slot, so they collide.

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, a hash function h is used 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 $\rightarrow$ {0,1, . . . , 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 1.2 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.

Two keys may hash to the same slot--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 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 1.4 introduces an alternative method for resolving collisions, called open addressing.

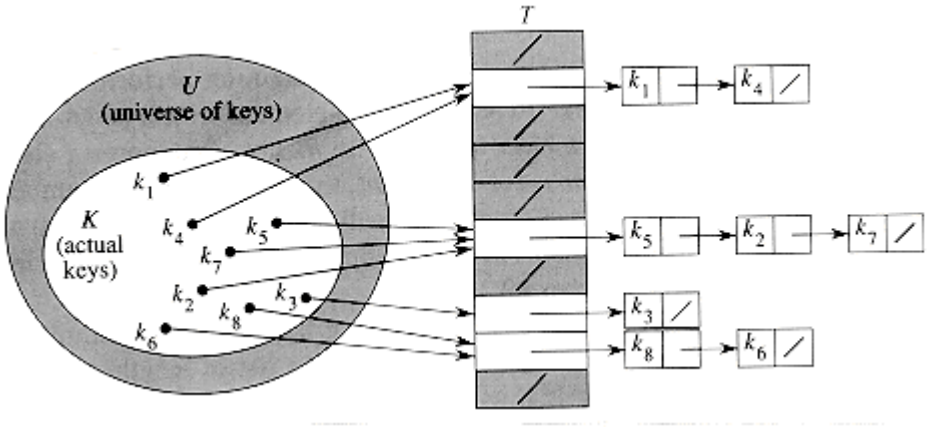

Figure 1.3 Collision resolution by chaining. Each hash-table slot T[j] contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

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 1.3. 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.

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(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(key[x])]

The worst-case running time for insertion is O(1). For searching, the worst-case running time is proportional to the length of the list; we shall analyze this more closely below. Deletion of an element x can be accomplished in O(1) time if the lists are doubly linked. (If the lists are singly linked, we must first find x in the list T[h(key[x])], so that the next link of x's predecessor can be properly set to slice x out; in this case, deletion and searching 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; that is, we imagine α staying fixed as n and m go to infinity. (Note that α can be less than, equal to, or greater than l .)

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.

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 1.3 discusses these issues, but for now we shall assume that any given element is equally likely(having equal probability) 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.

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 of the list T[h(k)]. Setting aside the O(1) time required to compute the hash function and 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 1.1

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

Proof Under the assumption of simple uniform hashing, any key k is equally likely to hash to any of the m slots. The average time to search unsuccessfully for a key k is thus the average time to search to the end of one of the m lists. The average length of such a list is the load factor α = n/m. 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 $\Theta(1+\alpha)$.

Theorem 1.2

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 key being searched for is equally likely to be any of the n keys stored in the table. We also assume that the CHAINED-HASH-INSERT procedure inserts a new element at the end of the list instead of the front. The expected number of elements examined during a successful search is 1 more than the number of elements examined when the sought-for element was inserted (since every new element goes at the end of the list). To find the expected number of elements examined, we therefore take the average, over the n items in the table, of 1 plus the expected length of the list to which the ith element is added. The expected length of that list is (i- 1)/m, and so the expected number of elements examined in a successful search is

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)\\ &= 1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right)\\ &= 1+\frac{\alpha}{2}-\frac{1}{2m}.\end{aligned}$$

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

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, α = n/m = O(m)/m = O(1). Thus, searching takes constant time on average. Since insertion takes O(l) worst-case time and deletion takes O(l) worst-case time when the lists are doubly linked, all dictionary operations can be supported in O(l) time on average.

Reference :

Text Book-T. H. Cormen, C. E. Leiserson , R. L. Rivest and C. Stein : Introduction to Algorithms, Third Edition ,The MIT Press Cambridge, Massachusetts London, England .