Hash Table

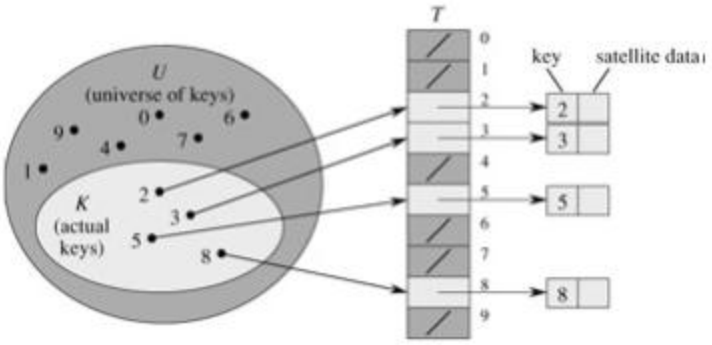
Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler for a computer language maintains a symbol table, in which the keys of elements are arbitrary character strings that correspond to identifiers in the language. A hash 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. In the next section, we shall discuss direct addressing in 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.

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 $U = \{0, 1, \ldots, m - 1\}$, 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 . Below figure 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.



Above figure explains 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.

The dictionary operations are trivial to implement.

```
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.

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,

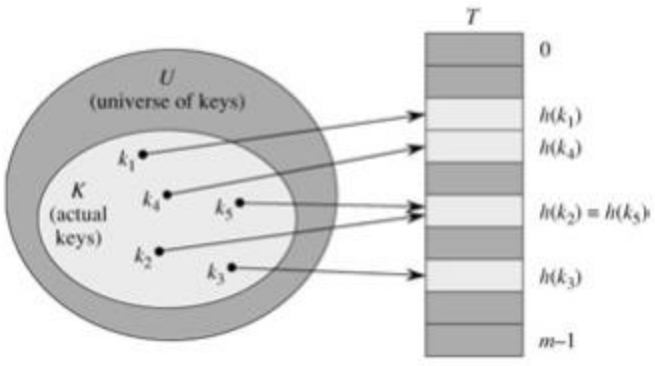
whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

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 Θ(|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 m - 1]: h : U → {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. The basic idea is illustrated in below figure. 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. Keys k2 and k5 map to the same slot, so they collide.



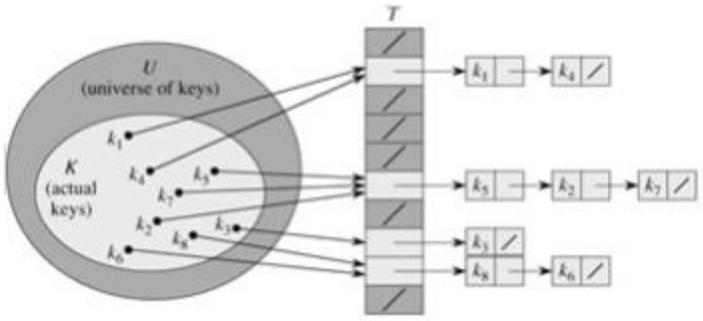
There is one hitch: two keys may hash to the same slot. We call this situation a collision. Fortunately, there are effective techniques for resolving the conflict created by collisions. Of course, the ideal solution would be to avoid collisions altogether. We might try to achieve this goal by choosing 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. There is 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 below figure. 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. Each hash-table slot T[j] contains a linked list of all the keys whose hash value is j. For example, h(k1) = h(k4) and h(k5) = h(k2) = h(k7). 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). 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-HASHDELETE 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(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.)

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 $0 \le k < 1$, 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 later) 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 propertiesthan 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. Universal hashing often provides the desired properties.

Source:

http://www.learnalgorithms.in/#