11. Hash Tables

Many applications require a **dynamic set** that supports only the **directory operations** `INSERT`, `SEARCH` and `DELETE`.

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 the array in $O(1)$ time, independent of the size of array n.

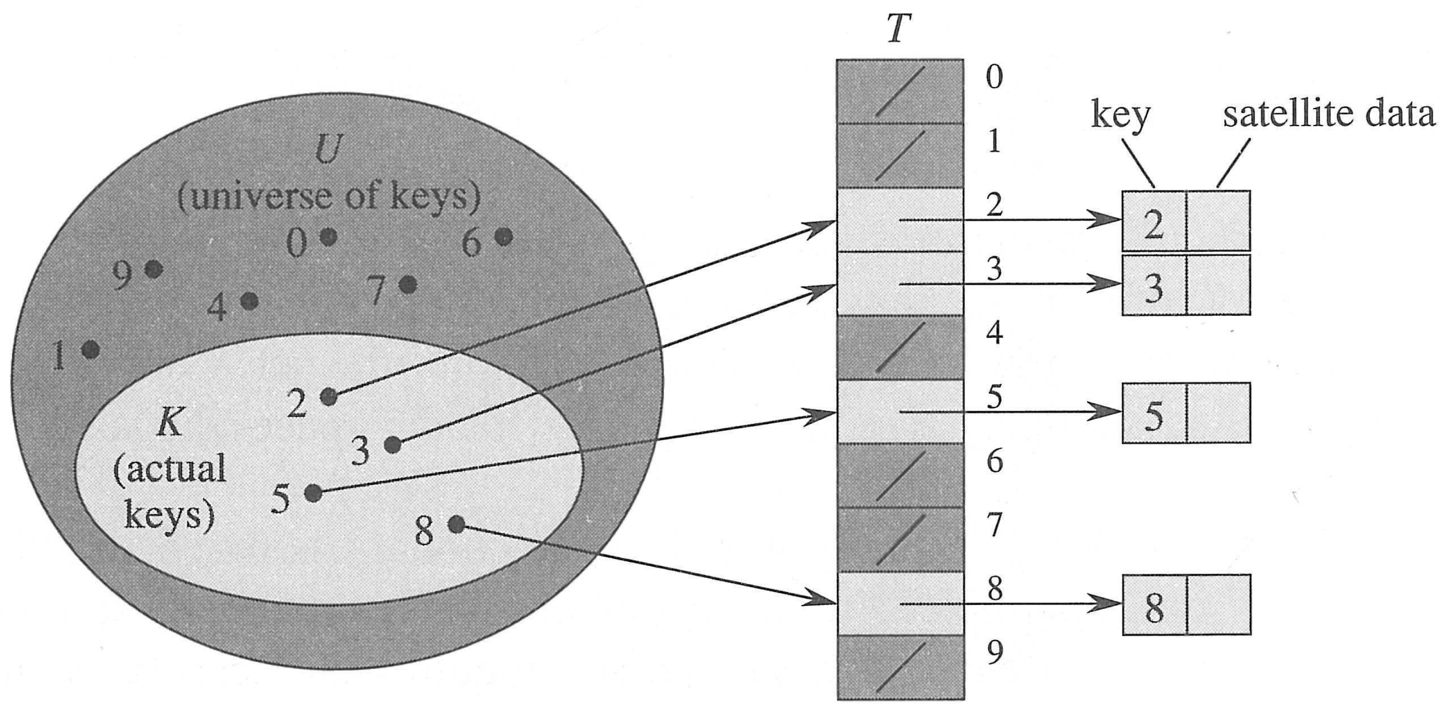
Direct addressing is a simple technique that works well when the universe U of the keys (all possible values of k) is reasonably small,

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

where

- m is not too large
- no two elements share the same key.

11.1. Direct-address tables



Direct-Addressing (Cormen et al., p254)

11.1 Operations on direct-address tables

To represent a dynamic set consisting of insertion, deletion, and searching, we use a direct-address table, denoted by $T[0..m-1]$, in which each position, or **slot**, corresponds to a key in the universe U.

- DIRECT_ADDRESS_SEARCH(T,k)
  return $T[k]$

- DIRECT_ADDRESS_INSERT(T,x)
  $T[key[x]] \leftarrow x$

- DIRECT_ADDRESS_DELETE(T,x)
  $T[key[x]] \leftarrow NIL$

11.2 Hash tables

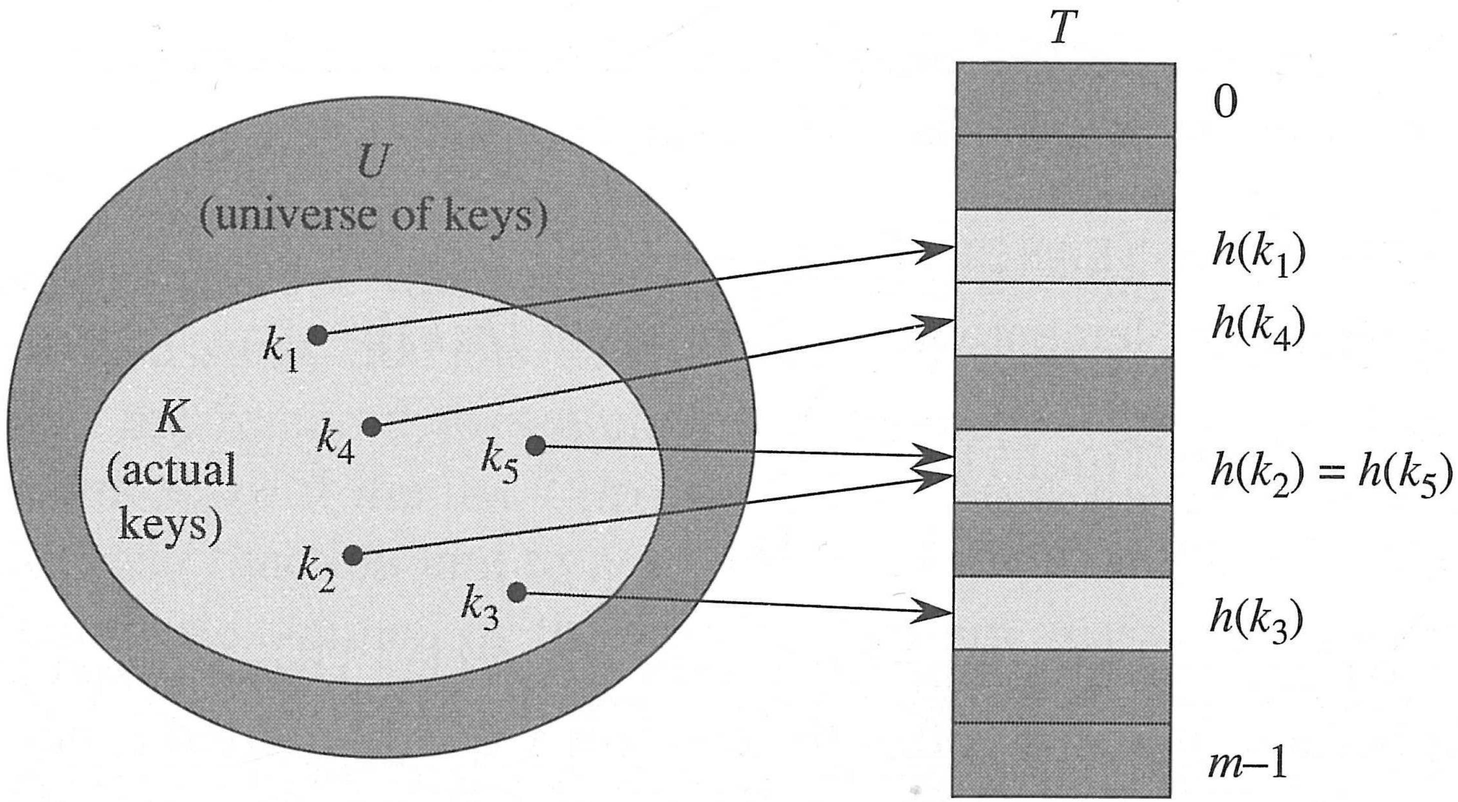
With hashing, the element is stored in slot $h(k)$, i.e., we use a **hash function** h to compute the slot for the element using key k, where h maps the universe U of keys into the slots of a **hash table** $T[0..m-1]$.

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

- 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.

Notice that with direct addressing, an element with key k is stored in slot k, which is a very special hash table.

11.2 Hash tables



Using a hash function (Cormen et al., p256)

11.2 Collision in Hash tables

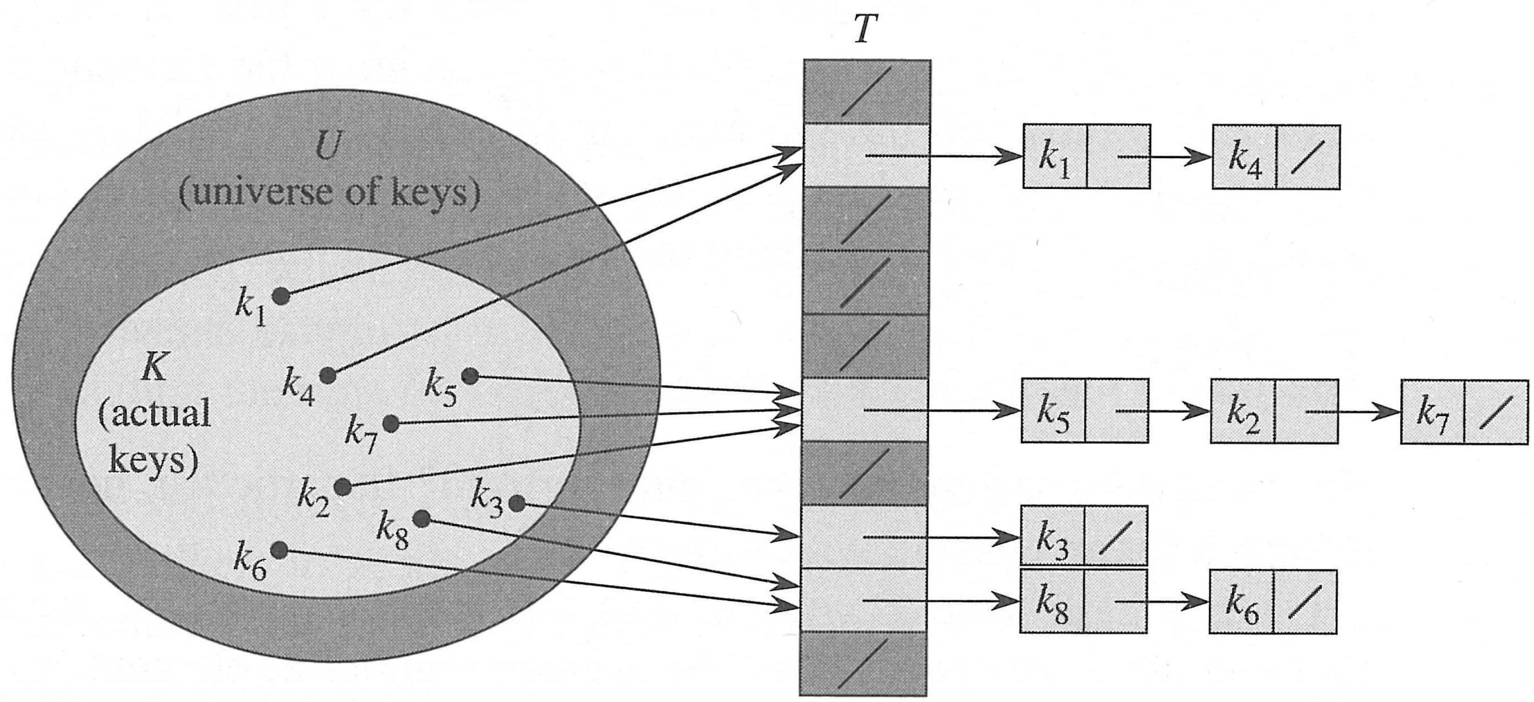
The drawback of hash tables is **the collision** when two different keys are mapped to **the same slot**.

Resolving collisions are a key issue in the design of hash functions.

One effective way to resolve collisions is called chaining and works as follows:

Put all the elements that hash to the same slot in a linked list.

11.2 Collision in Hash tables



Avoiding collisions using linked lists (Cormen et al., p257)

11.2 Operations on hash tables

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

- CHAINED_HASH_SEARCH(T, k)
  search for an element with key k in list $T[h(k)]$
- CHAINED_HASH_INSERT(T, x)
  insert x at the head of list $T[h(key[x])]$ (**Why?**)
- CHAINED_HASH_DELETE(T, x)
  delete x from the list $T[h(key[x])]$ (**How to delete x from the list?**)

The worst case behavior of hashing with chaining takes $O(n)$ time when searching or deleting an element from the table.

11.2 Analysis of simple uniform hashing with chaining

Given a hash table with m slots that stores n elements, the **load factor**

$$\alpha = n/m.$$

A simple uniform hashing assumes that any given element is equally likely to hash into any of the m slots, **independently of where any other element has hashed to**. The average behavior of hashing under this assumption is much better, which is $\Theta(1+\alpha)$.

Let the hash table contain m slots. For $j = 0, \ldots, m-1$, denote the length of list $T[j]$ by n_j,

so that $n = n_0 + n_1 + \ldots + n_{m-1}$, and the average value of n_j is $E[n_j] = \alpha = n/m$.

What is the relationship of the load factor α with th time of searching/deletion of an element?

11.2 Analysis of simple uniform hashing with chaining (cont.)

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

Case 1: Unsuccessful search for key k:

The list for hash value $h(k)$ has to traversed. Its expected length is $E[n_j] = \alpha = n/m$.

Case 2: Successful search for key k:

Let $k_i = key[x_i]$. For keys k_i and k_j define $X_{ij} = I\{h(k_i) = h(k_j)\}$ as a random variable. $Pr\{h(k_i) = h(k_j)\} = 1/m$. Thus, $E[X_{ij}] = 1/m$.
Assume that key k_i is hashed to a slot $h(k_i)$, when we retrive the linked list in which key k_i is contained from the head of the list, we can find all keys in front of key k_i must be k_j with $j > i$. In other words, the amount of time spent on this linked list to

identify k_i is proportional to the number of keys before it, while the probability of a key k_j with $j > i$ in front of key k_i is $1/m$, thus, we have

$$E[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] = 1+\frac{n-1}{2m} = 1+\alpha/2-\alpha/2n = \Theta(1+\alpha)$$