

# Chapter 11: Hash Tables

To search an element in a hash table:

Worst-case:  $\Theta(n)$  (no better than a linked list).

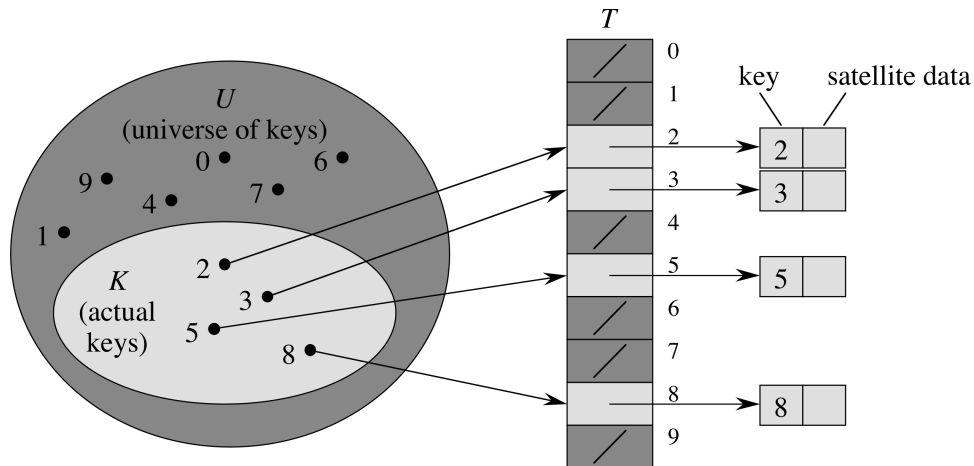
Average-case:  $O(1)$ .

## Direct-address Tables

- Each element to be stored has a unique key.
- Suitable for applications with small set  $U$  (the set of all possible keys).
- Suppose  $U = \{0, 1, \dots, m\}$ . To store a dynamic set of elements, a direct-address table  $T[0 \dots m - 1]$  is allocated, where

$$T[i] = \begin{cases} x & \text{if the element with key } i \text{ is stored, where } x \text{ points to the element} \\ \text{nil} & \text{otherwise} \end{cases}$$

Ex:  $U = \{0, 1, \dots, 9\}$  and the set of elements (keys) stored =  $\{4, 7, 9\}$ .



Direct-Address-Search( $T, k$ )

1. return  $T(k)$  //  $O(1)$

Direct-Address-Insert( $T, x$ )

<<<<<<< HEAD

1.  $T[x.key] = x$  //  $O(1)$

Direct-Address-Delete( $T, x$ )

1.  $T[x.key] = \text{nil}$  //  $O(1)$

```

=====
1. T[key[x]] = x                // O(1)

Direct-Address-Delete(T, x)
1. T[key[x]] = nil              // O(1)
>>>>>> fac36428d911c42afb28c4b9d34feb81c1776b9c

```

## Hash Tables

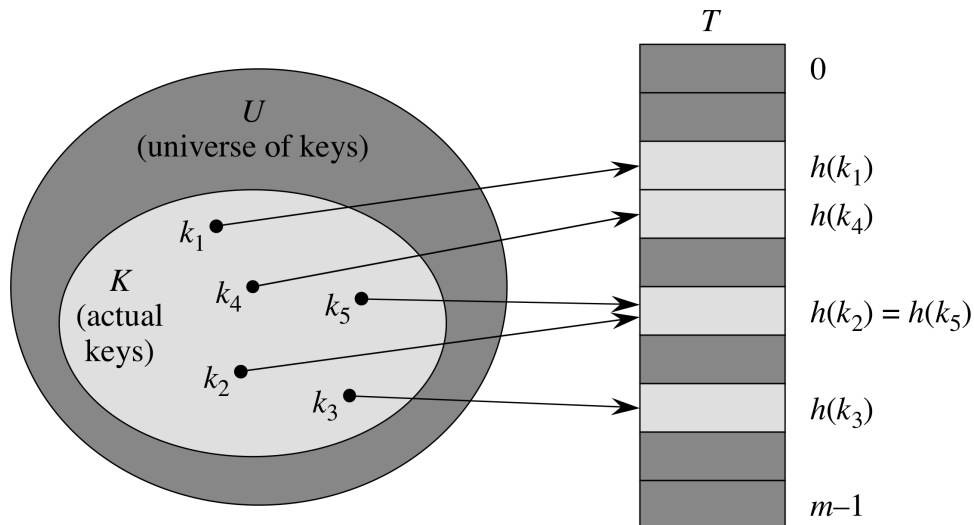
- It's impractical for direct addressing if  $U$  is large.  
Ex: If  $U$  is the set of all possible SSN, then  $T$  requires  $10^9$  entries.
- We can reduce the memory requirement to  $\Theta(n)$  and still have  $O(1)$  average searching time by hashing technique, where  $n$  is the number of elements stored.

Direct-addressing: an element with key  $k$  is stored in entry  $k$ .

Hashing : an element with key  $k$  is stored in entry  $h(k)$ ,  
where  $h$ : hash function and  $h(k)$ : hash value.

For a **hash table**  $T[0..m-1]$ , the **hash function**  $h: U \rightarrow \{0, 1, \dots, m-1\}$ .

- The hashing technique is for applications that  $|U| \gg m$  (the size of table  $T$ ).  
 $\implies$  **collisions** may occur (two keys  $k_1, k_2$  that  $h(k_1) = h(k_2)$ ).



### Solution for collision:

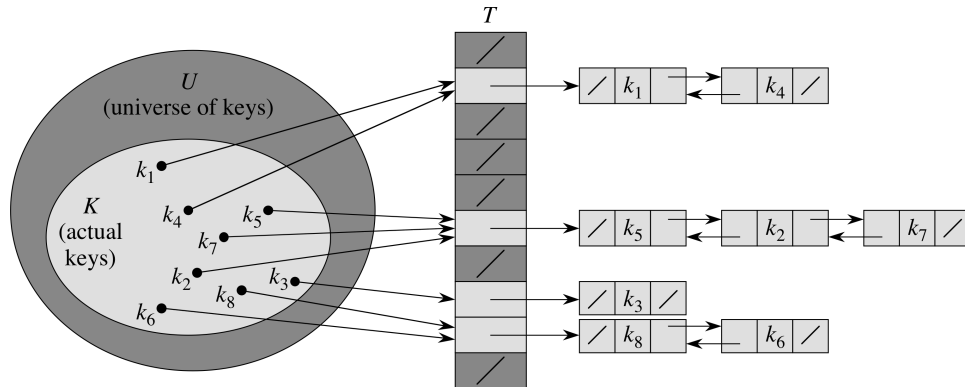
- avoid the collisions altogether: impossible.
- choose  $h$  to be “random” to minimize the # of collisions.

Two topics to research on:

1. 1. What is a good choice of  $h$ ?
2. 2. How to resolve the collisions?

## Collision resolution by chaining

We put all elements colliding on one entry into a linked list. For example:



Chained-Hash-Insert( $T, x$ )

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

Chained-Hash-Search( $T, k$ )

1. search for an element with key  $k$  in list  $T[h(k)]$

**Average time:**  $\Theta(1 + \alpha)$ , where  $\alpha = n/m$  is the load factor of the table  $T$ . Based on *Theorem 11.1* and *Theorem 11.2* in the textbook. We will skip the proofs of those theorems though for this class.

Chained-Hash-Delete( $T, x$ )

1. delete  $x$  from the list  $T[h(x.key)]$  //  $O(1)$ : if doubly linked list.  
// Same running time as search  
// if singly linked list.

## Hash Functions

- Good hash function: **simple uniform hashing**.  
Each key is equally likely to hash to any of the  $m$  entries.
- Assumption: Let the domain of keys is the set of natural numbers.  
If the keys are not numbers (e.g., strings), they should be mapped to numbers before hashing.

## The division method

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

- the division method is almost simple uniform.
- $m$  should not be a power of 2 (if  $m = 2^p$ , then  $h(k)$  is the  $p$  low-order bits of  $k$ . It is better to make hash function depends on all bits of the key).
- Good choice for  $m$ : primes that are not too close to exact powers of 2.

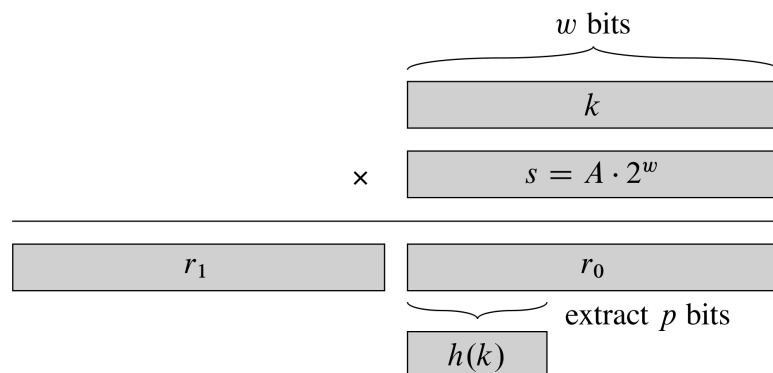
*Example:* To hold 2000 keys and 3 elements examined in average for an unsuccessful search. The collision is resolved by chaining. What the table size  $m$  should be?

Sol:  $m = 701$ , since 701 is a prime and  $701 \simeq 2000/\alpha$ , and also 701 is not close to any power of 2.

## Multiplication method

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

- The value of  $m$  isn't critical so we could choose to be a power of 2 ( $m = 2^p$  for some integer  $p$ ) to make the calculation of the hash function easier.



**Recommended Exercises:** 11.3-1, 11.3-2.

# Open Addressing

- Another collision resolution technique.
- Each entry in the table  $T$  can store at most one element, rather than a linked list of elements in chaining. Thus,  $\alpha \leq 1$ .
- To perform insertion, we successively examine, or probe, the hash table until we find an empty entry.
- The probe sequence (the sequence of entries examined) depends on the probing techniques used. There are 3 popular probing techniques.

## Pseudocode:

```
Hash-Search(T, k)
1. i = 0
2. repeat
3.     j = h(k, i)
4.     if T[j] == k
5.         return j
6.     else i = i+1
7. until T[j] == nil or i == m
8. return nil
```

```
Hash-Insert(T, k)
1. i = 0
2. repeat
3.     j = h(k, i)
3.     if T[j] == nil or DELETED
4.         T[j] = k
5.         return j
6.     else i = i+1
7. until i == m
8. error "hash table overflow"
```

```
//If an entry is occupied by an element but the element
//has been deleted later, we need to set a flag ``deleted'' to this entry.
```

```
Hash-Delete(T, k)
1. i = 0
2. repeat
```

```

3.     j = h(k, i)
3.     if T[j] == k
4.         T[j] = DELETED
5.         return j
6.     else i = i+1
7. until T[j] == nil or i == m
8. error  "k is not in the table"

```

**Linear Probing:** The probe sequence is:

$$h(k, i) = (h'(k) + i) \bmod m, \text{ for } i = 0, 1, \dots, m-1$$

That is,  $T[h'(k)], T[h'(k) + 1], \dots, T[m-1], T[0], T[1], \dots, T[h'(k) - 1]$ .

This technique suffers on the **primary clustering** problem: long runs of occupied entries.

- Why does linear probing usually results in primary clustering?
- What's wrong with the primary clustering?
- An empty slot preceded by  $i$  full slots gets filled next with probability  $(i+1)/m$ . So long runs tend to get longer, thus decreasing the average search time.

**Quadratic Probing:** The probe sequence is

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \text{ where } c_1, c_2 > 0, i = 0, 1, \dots, m-1$$

$c_1, c_2$  and  $m$  need to be chosen carefully to **fully utilize** the table  $T$  (see Problem 11.3).

- To fully utilize the table  $T$ , we mean that for given any key  $k$ , the probe sequence of  $k$  can check the entire table.

If two keys  $k_1, k_2$  have the same initial probe position, then they will have the same probe sequence. That is,  $h(k_1, 0) = h(k_2, 0)$  implies  $h(k_1, i) = h(k_2, i), \forall i$ .

This phenomenon leads to a milder form of clustering: **secondary clustering**.

**Double Hashing:** The probe sequence is

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m, \text{ for } i = 0, 1, \dots, m-1$$

- The probe sequence depends in two ways upon the key  $k$ . In this case, if two keys have the same initial probe position, they may not have the same probe sequence. Thus, double hashing does not suffer from secondary clustering.
- In order to fully utilize the table, given any  $k$ ,  $h_2(k)$  **must be always relatively prime to  $m$** .

If  $d = \gcd\{h_2(k), m\}$ , then only  $\frac{1}{d}$  th of table will be searched.

Two different approaches to make  $h_2(k)$  always relatively prime to  $m$ .

1. Let  $m = 2^p$ , where  $p$  is some positive integers.  
Design  $h_2$  so that it always produces an odd number.
2. Let  $m$  be a prime number.  
Design  $h_2$  so that it always produces a positive integer less than  $m$ .

Ex: Let  $m$  be a prime and let

$$\begin{cases} h_1(k) = k \bmod m \\ h_2(k) = 1 + (k \bmod m'), \text{ where } m' = m - 1 \text{ or } m - 2 \end{cases}$$

**Recommended Exercise:** 11.4-1, 11.4-2.

Analysis of hashing by chaining and open address hashing:

**Theorem 11.1** *For hashing by chaining, an unsuccessful search takes time  $\Theta(1 + \alpha)$  in average, under the simple uniform hashing assumption.*

**Theorem 11.2** *For hashing by chaining, a successful search takes time  $\Theta(1 + \alpha)$  in average, under the simple uniform hashing assumption.*

**Theorem 11.6** *For open-address hashing, the expected # of probes in an unsuccessful search is at most  $1/(1 - \alpha)$ , assuming uniform hashing.*

**Theorem 11.8** *For open-address hashing, the expected # of probes in a successful search is at most  $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ , assuming uniform hashing.*