Module 3
Hashing and Chaining

Seraching and Hash Tables

- Searching for an element in linked list : $\Theta(n)$ time
- Hash Table : effective data structure for implementing dictionaries
  - Separation into buckets
- Worst case time: Searching for an element in a hash table can take as long as searching for an element in a linked list $\Theta(n)$ time
- Hashing can change it to $O(1)$ time in best case scenario.
  - Which data structure can search in $O(1)$ time?

Implementing a **dictionary** with a **direct-address table**

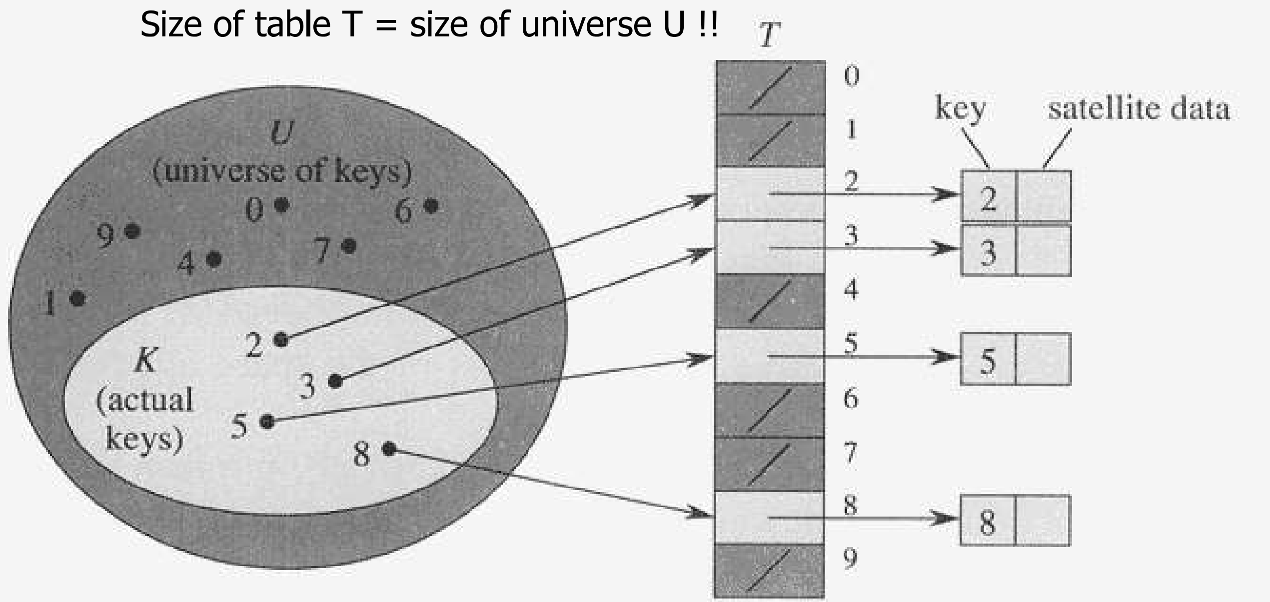


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,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

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

Implementing a **dictionary** with a **direct-address table**

Size of table T = size of universe U !!

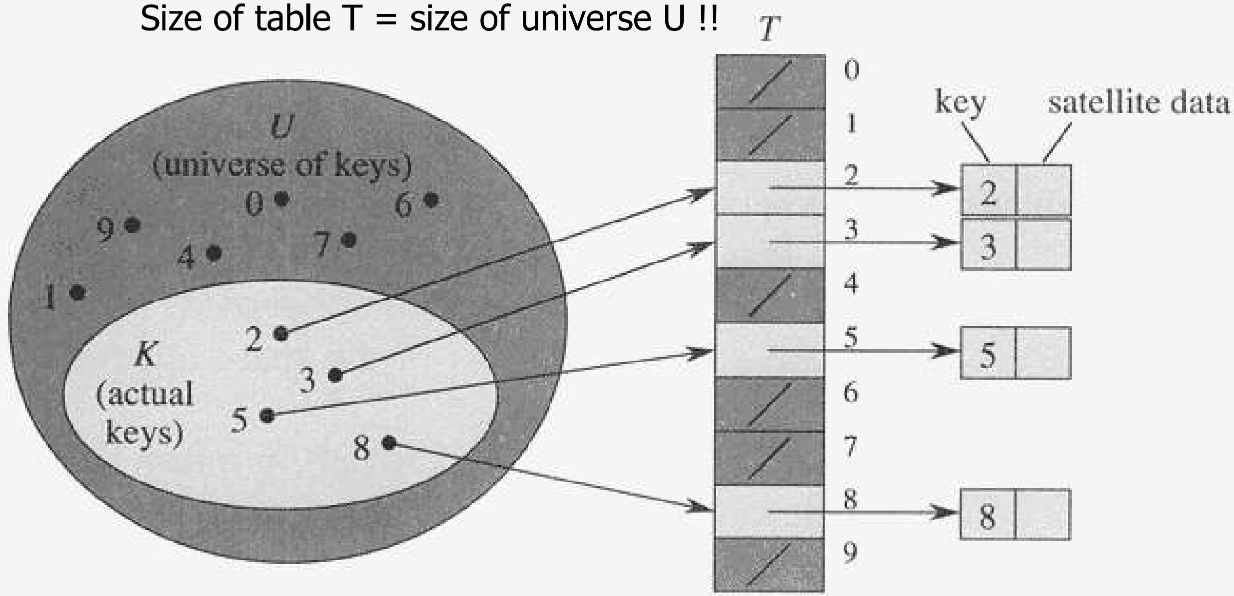


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,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

Cons: if the universe U is large, storing a table T of size [U] may be impractical

Hashing

- With **direct addressing**, an element with key k is stored in slot k.
- With **hashing**, this element is stored in slot h(k)
- 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[0,1...,m-1]

    where h:U --> {0,1,..,m-1} and size of m is much less than [U]

Reducing the size of the table using a **hash function** to map keys to a **hash table**

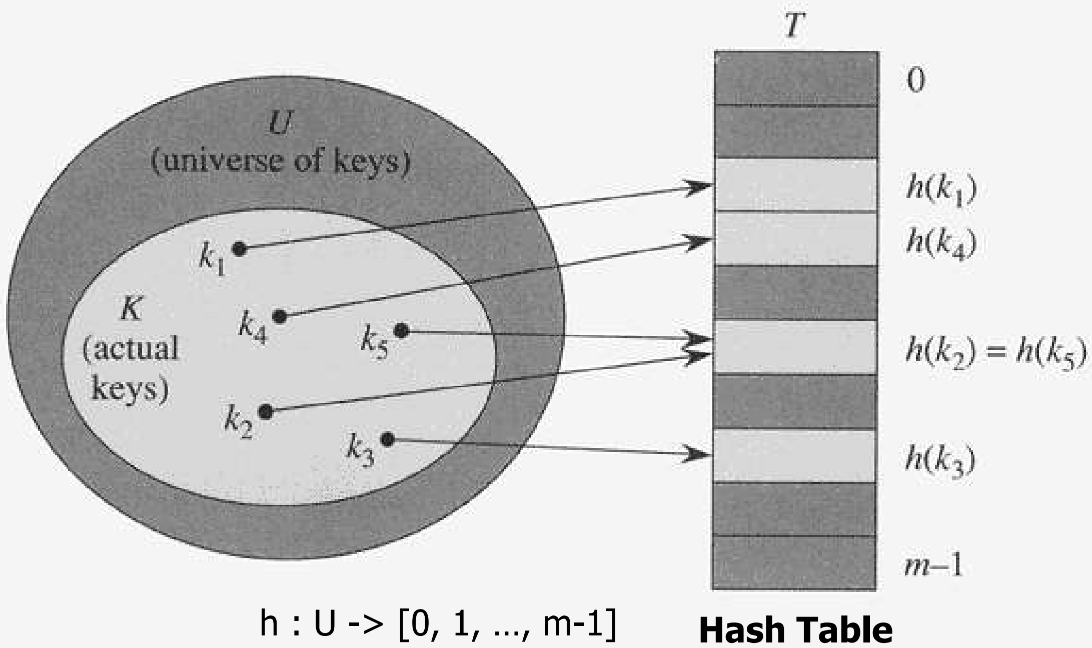


Figure 11.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

Hashing Example 1

- key space U = integers
- TableSize m = 10
- **h**(K) = K mod 10
- **Insert**: 7, 18, 41, 94
- Where do we insert each of them?

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing Example 2

- key space = integers
- TableSize = 6

- **h**(K) = K mod 6

- **Insert**: 7, 18, 41, 34

OR

- **Insert**: 7, 18, 41, 37

0	
1	
2	
3	
4	
5	

Reducing the size of the table using a **hash function** to map keys to a **hash table**

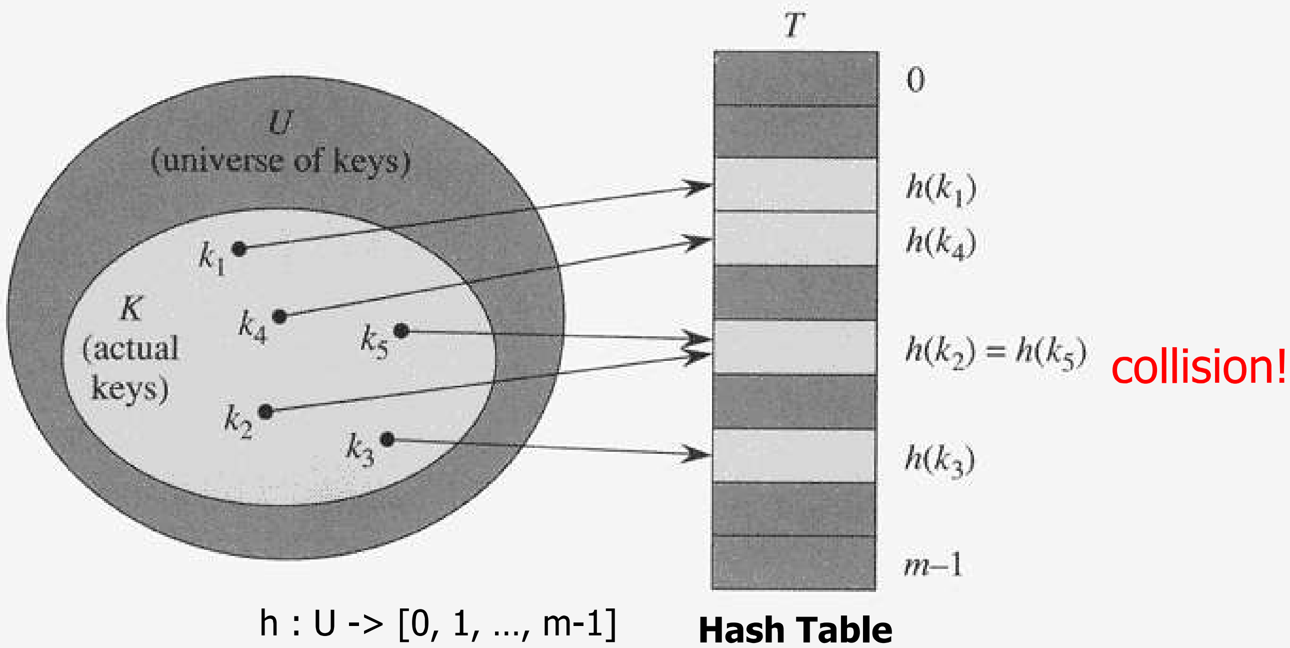


Figure 11.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

Collision resolution by chaining

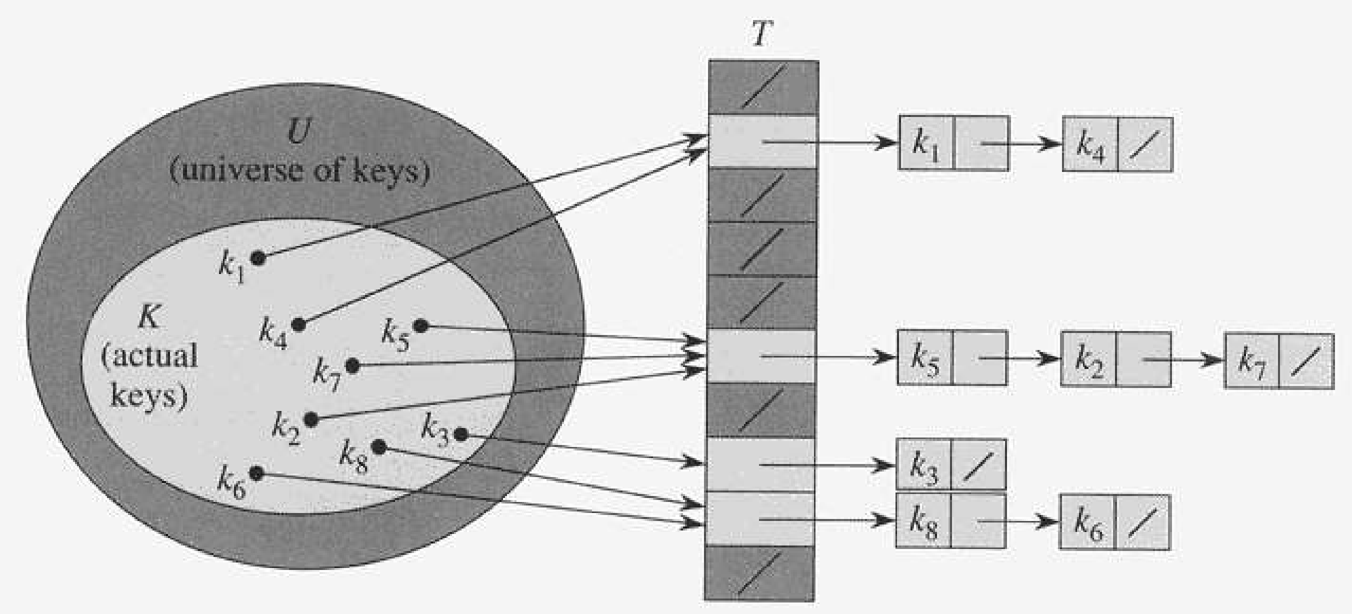


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Load factor α = # elements in table n / # slots in table m
= average # elements in a chain

CHAINED-HASH-INSERT(T, x) insert x at the head of the 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])]

What are worst-case times?!

Universal hashing

- A fixed hash function is vulnerable to malicious distributions (e.g., so that all keys hash to the same slot!)
- Universal hashing consists of choosing a hash function *randomly* from some fixed *set* of hash functions *independent* of the keys.
- Therefore, universal hashing is (probabilistically) immune to bad distributions (just like randomized quicksort is probabilistically immune to a bad input).

Collision Resolution

Collision: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

1. Separate Chaining – Open hashing
2. Open Addressing (linear probing, quadratic probing, double hashing) – Closed Hashing

1. Separate Chaining: How big should the hash table be?

- For Separate Chaining:
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - tableSize = 10
    data hashes to 0, 3, <u>0</u>, 5, 1, <u>0</u>, <u>0</u>
  - tableSize = 11
    data hashes to 10, 9, 5, 0, 2, <u>9</u>, 7

Real-life data tends to have a pattern

Being a multiple of 11 is usually *not* the pattern ☺

2. a) Open Addressing - Linear Probing

$f(i) = i$

- Probe sequence:

  0^{th} probe = $h(k) \bmod TableSize$

  1^{th} probe = $(h(k) + 1) \bmod TableSize$

  2^{th} probe = $(h(k) + 2) \bmod TableSize$

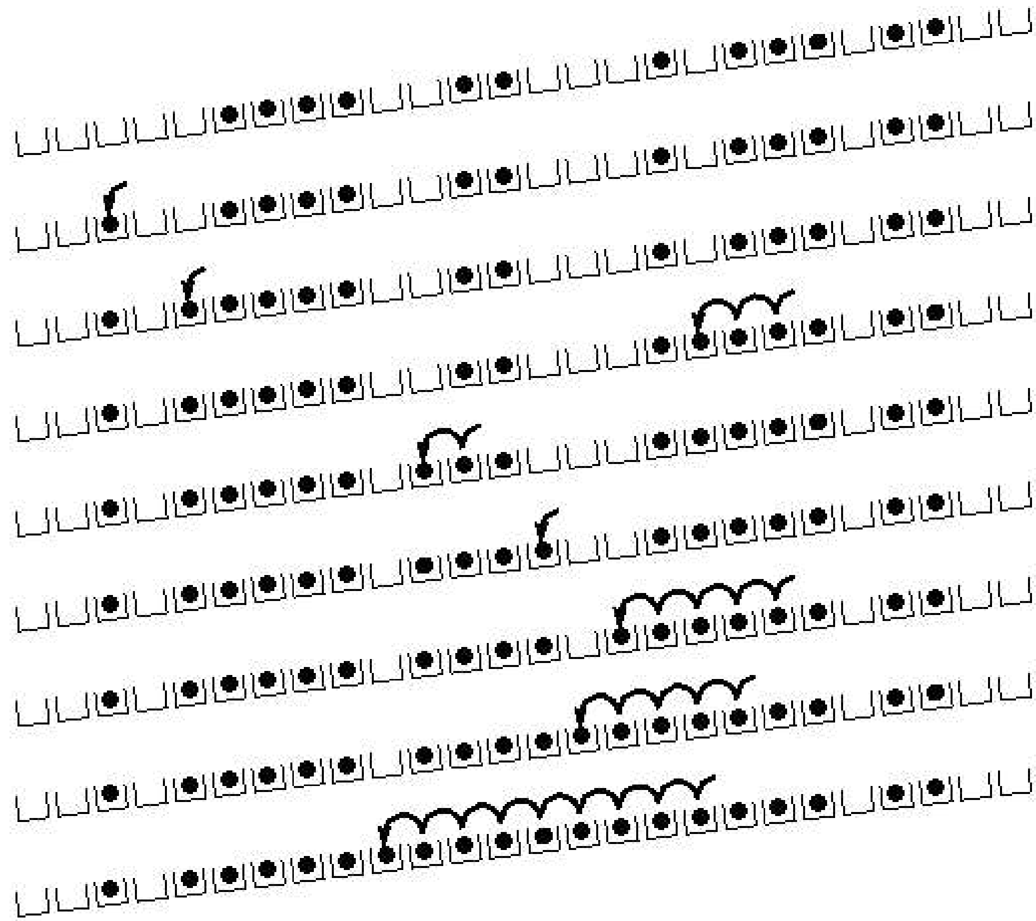
  . . .

  i^{th} probe = $(h(k) + i) \bmod TableSize$

2a) Open Addressing - Linear Probing

- works pretty well for an empty table and gets worse as the table fills up.

- primary clustering.



2. b) Quadratic Probing

- Add a function of I to the original hash value to resolve the collision.
- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 4) mod TableSize

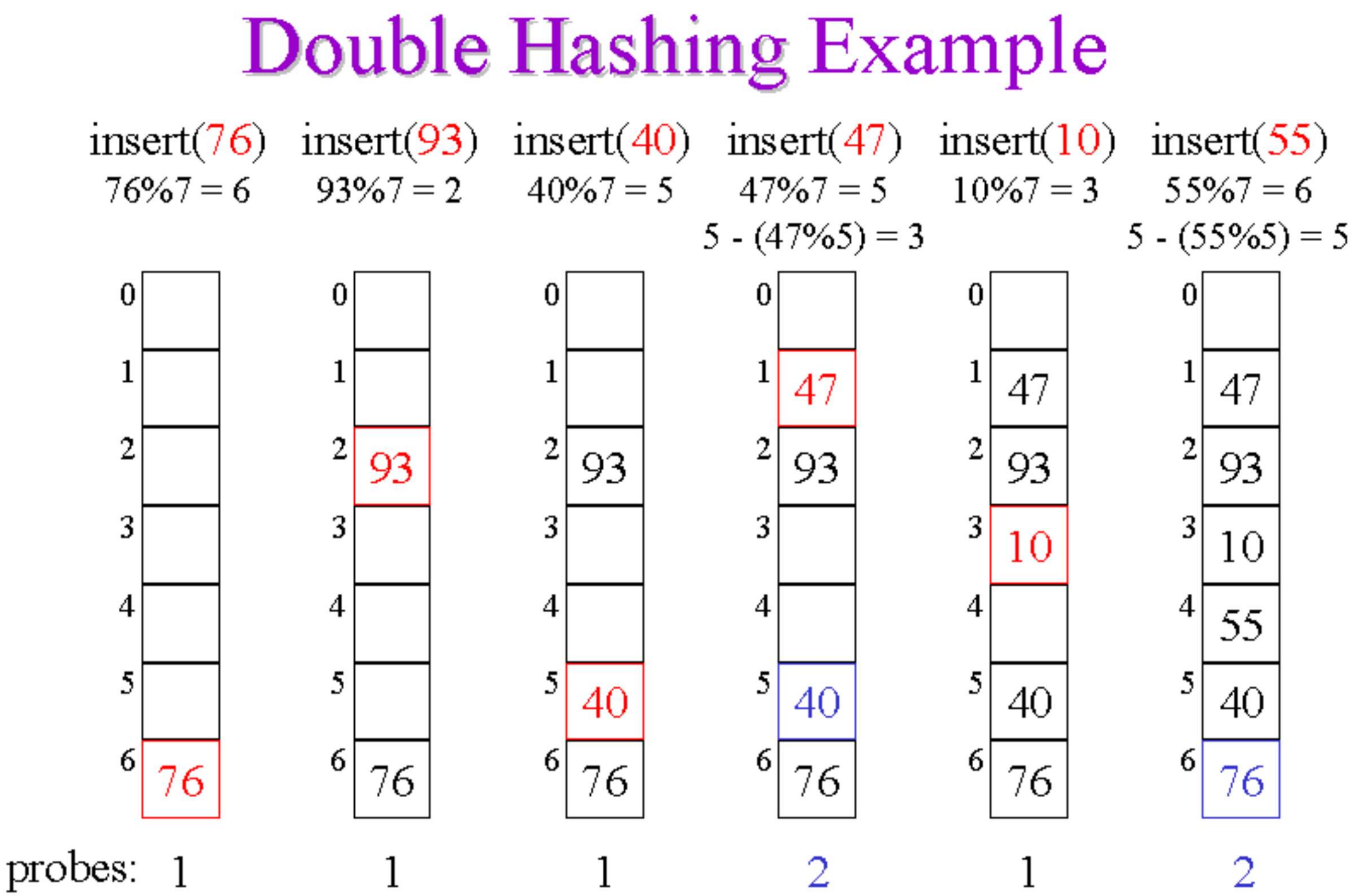
  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

  i^{th} probe = (h(k) + i^2) mod TableSize

Less likely to encounter Primary Clustering

2 c) Double Hashing



2 c) Double Hashing

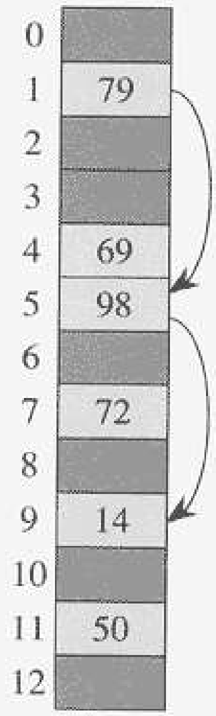


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Perfect Hashing

- If the set of keys is *static* (e.g., a set of reserved words in a programming language), hashing can be used to obtain excellent *worst-case* performance.
- A hashing technique is called **perfect hashing** if the worst-case time for a search is O(1).
- A *two-level* scheme is used to implement perfect hashing with universal hashing used at each level.
  - The first level is same as for hashing with chaining: n keys are hashed into m = n slots using a hash fn. h from a universal collection.
  - At the next level though, instead of chaining keys that hash to the same slot j, we use a small *secondary hash table* S_j with an associated hash fn. h_j. By choosing h_j appropriately one can guarantee that there are *no* collisions at the secondary level and that the total space used for all the hash tables is O(n).

Perfect Hashing

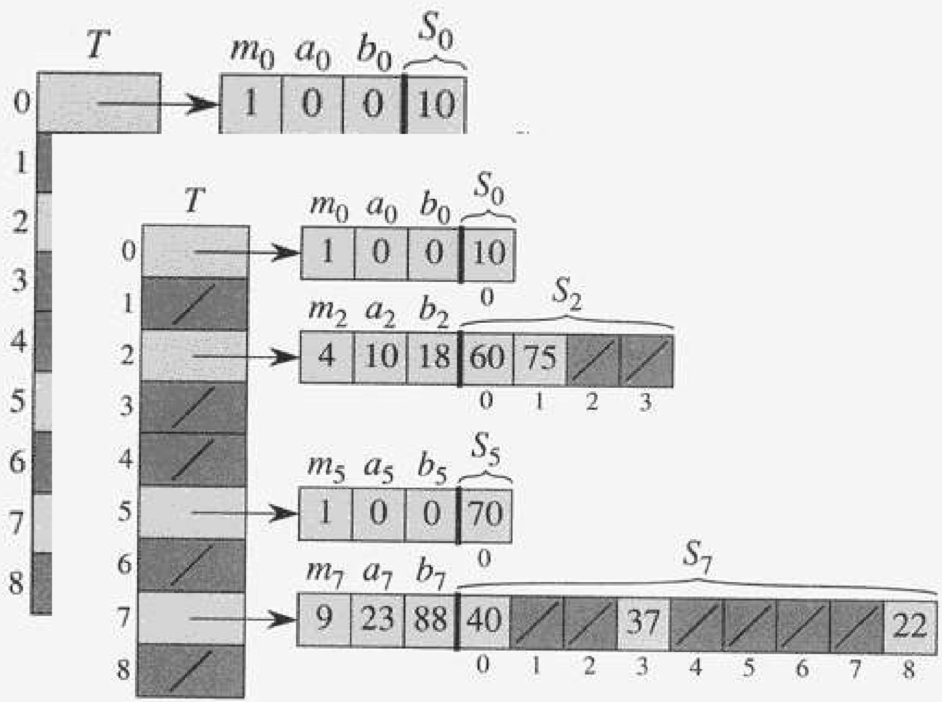


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 60, 70, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is m_j, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 1$, key 75 is stored in slot 1 of secondary hash table S_2. There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.