Hashing

adapted from David Luebke

Review: Linked Lists

- Think about a linked list as a structure for dynamic sets. What is the running time of:
  - **`Min()`** *and* **`Max()`** *?*
  - **`Successor()`** *?*
  - **`Delete()`** *?*
    - *How can we make this O(1)?*
  - **`Predecessor()`** *?*
  - **`Search()`** *?*
  - **`Insert()`** *?*

These all take O(1) time in a doubly linked list. Can you think of a way to do these in O(1) time in a red-black tree?

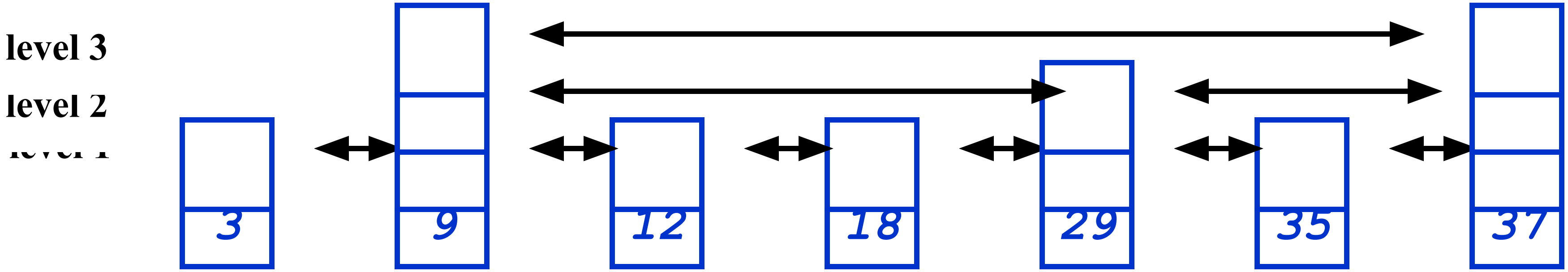
A: *threaded* red-black tree w/ doubly linked list connecting nodes in sorted order

Goal: make these O(lg n) time in a linked-list setting

Idea: keep several levels of linked lists, with high-level lists skipping some low-level items

Skip Lists (not included)

- The basic idea:



- Keep a doubly-linked list of elements
  - Min, max, successor, predecessor: O(1) time
  - Delete is O(1) time, Insert is O(1)+Search time
- During insert, add each level-*i* element to level *i*+1 with probability *p* (e.g., *p* = 1/2 or *p* = 1/4)

Skip List Search (not included)

- To search for an element with a given key:
  - Find location in top list
    - Top list has O(1) elements with high probability
    - Location in this list defines a range of items in next list
  - Drop down a level and recurse
- O(1) time per level on average
- O(lg n) levels with high probability
- Total time: O(lg n)

Skip List Insert (not included)

- Skip list insert: analysis
  - Do a search for that key
  - Insert element in bottom-level list
  - With probability p, recurse to insert in next level
  - Expected number of lists = $1 + p + p^2 + \ldots =$???

    $= 1/(1-p) = O(1)$ if p is constant
  - Total time = Search + $O(1) = O(\lg n)$ expected
- Skip list delete: $O(1)$

Skip Lists (not included)

- O(1) expected time for most operations
- O(lg n) expected time for insert
- O(n^2) time worst case (*Why?*)
  - But random, so no particular order of insertion evokes worst-case behavior
- O(n) expected storage requirements (*Why?*)
- Easy to code

Hashing Tables

- Motivation: symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
  - We typically don’t care about sorted order

Hash Tables

- More formally:
  - Given a table T and a record x, with key (= symbol) and satellite data, we need to support:
    - Insert (T, x)
    - Delete (T, x)
    - Search(T, x)
  - We want these to be fast, but don't care about sorting the records
- The structure we will use is a *hash table*
  - Supports all the above in O(1) expected time!

Hashing: Keys

- In the following discussions we will consider all keys to be (possibly large) natural numbers
  - *How can we convert floats to natural numbers for hashing purposes?*
  - *How can we convert ASCII strings to natural numbers for hashing purposes?*

Direct Addressing

- Suppose:
  - The range of keys is 0..m-1
  - Keys are distinct
- The idea:
  - Set up an array T[0..m-1] in which
    - T[i] = x if $x \in T$ and key[x] = i
    - T[i] = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time!
    - *So what's the problem?*

Direct Addressing

- .

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

Direct Addressing

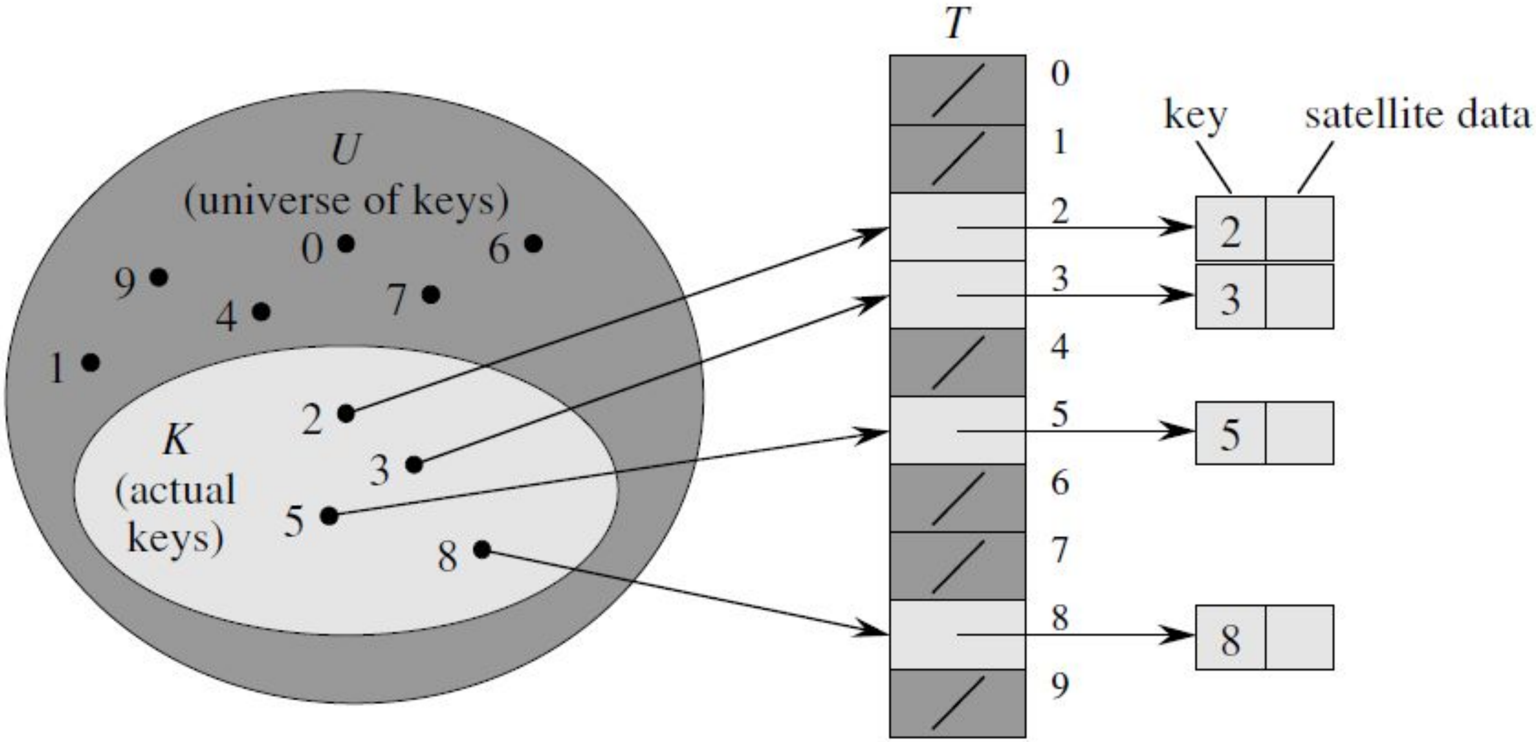


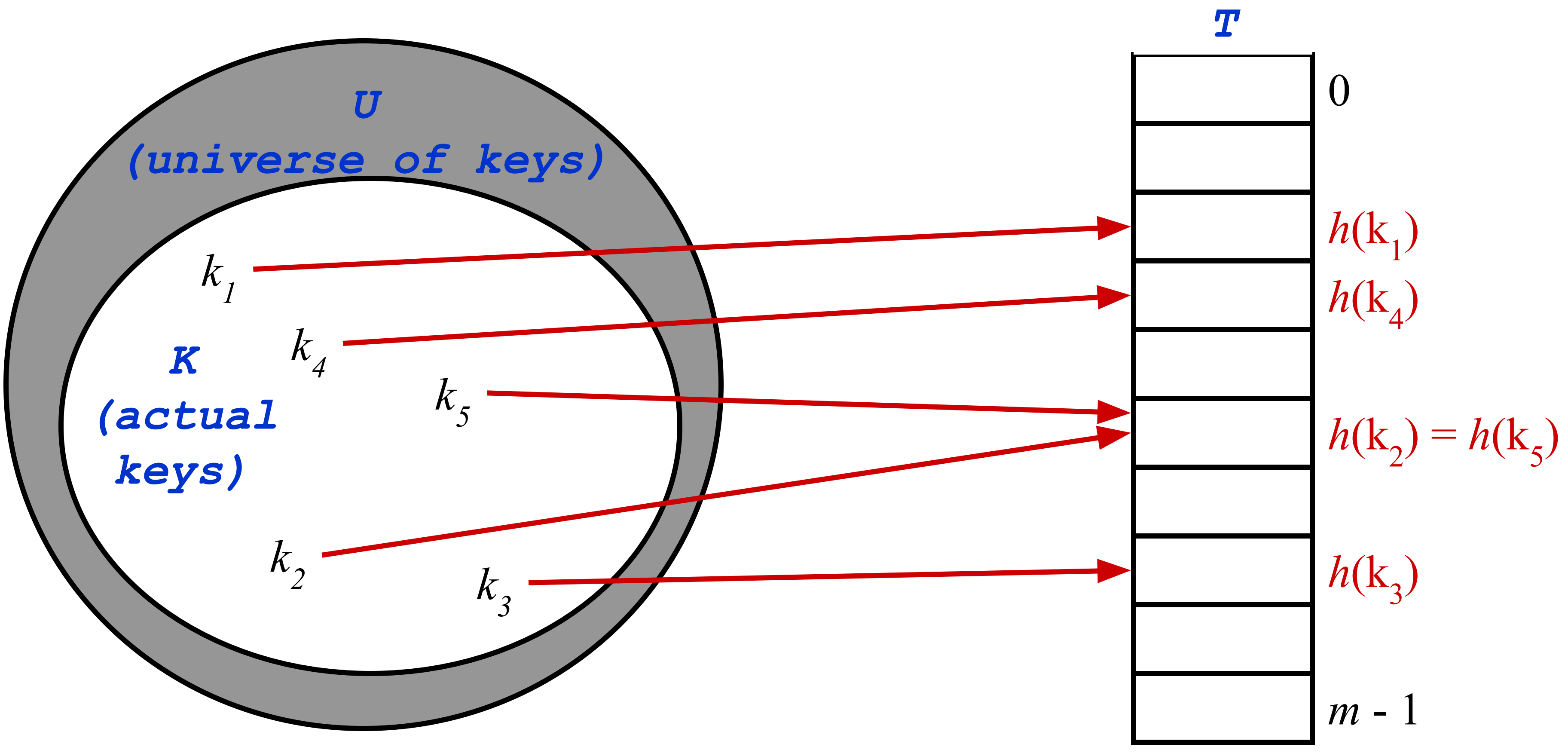
Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,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

The Problem With Direct Addressing

- Direct addressing works well when the range m of keys is relatively small
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range $0..m-1$
- This mapping is called a *hash function*

Hash Functions

- Next problem: *collision*

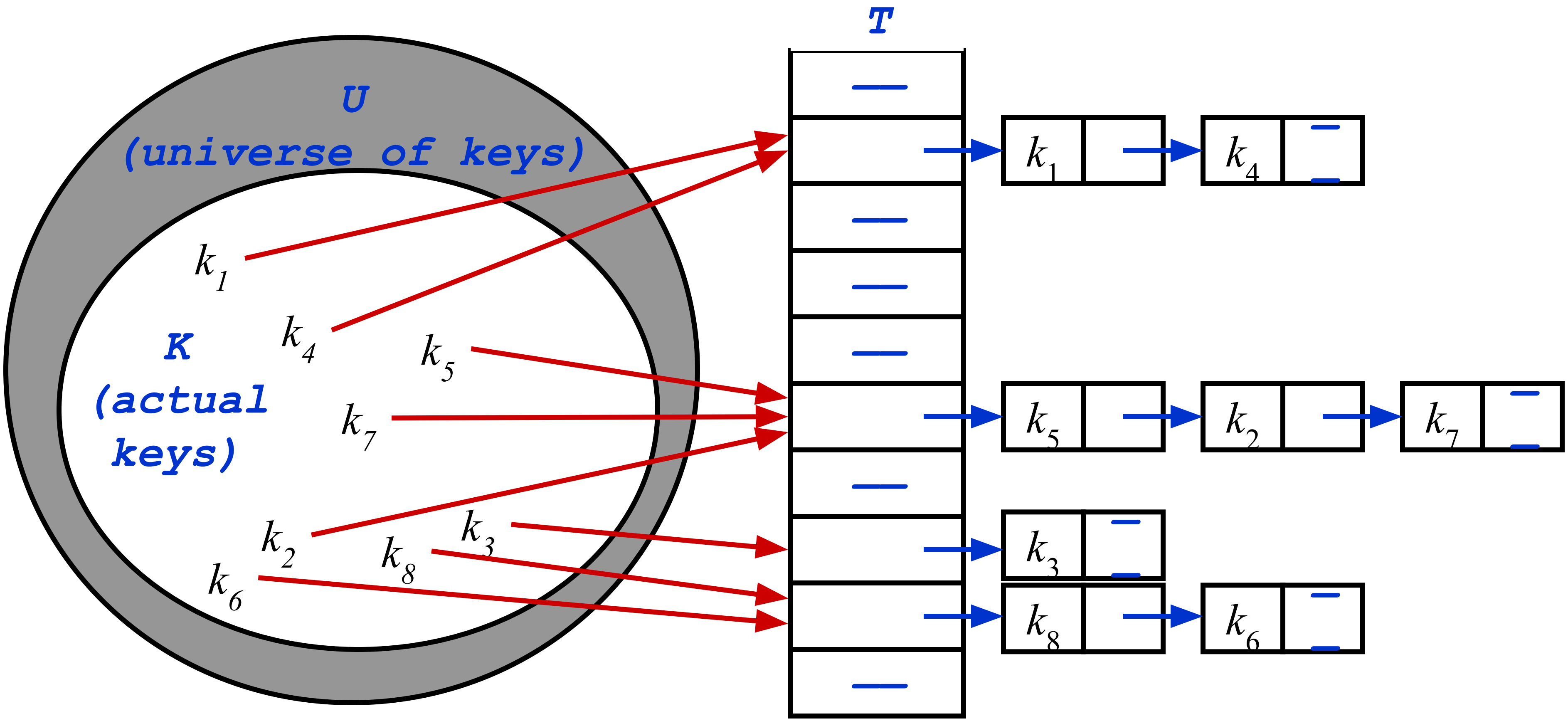


Resolving Collisions

- *How can we solve the problem of collisions?*
- Avoid collisions
- Solution 1: *chaining*
- Solution 2: *open addressing*

Chaining

- Chaining puts elements that hash to the same slot in a linked list:



Chaining

CHAINED-HASH-INSERT(T, x)

insert x at the head of list $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)

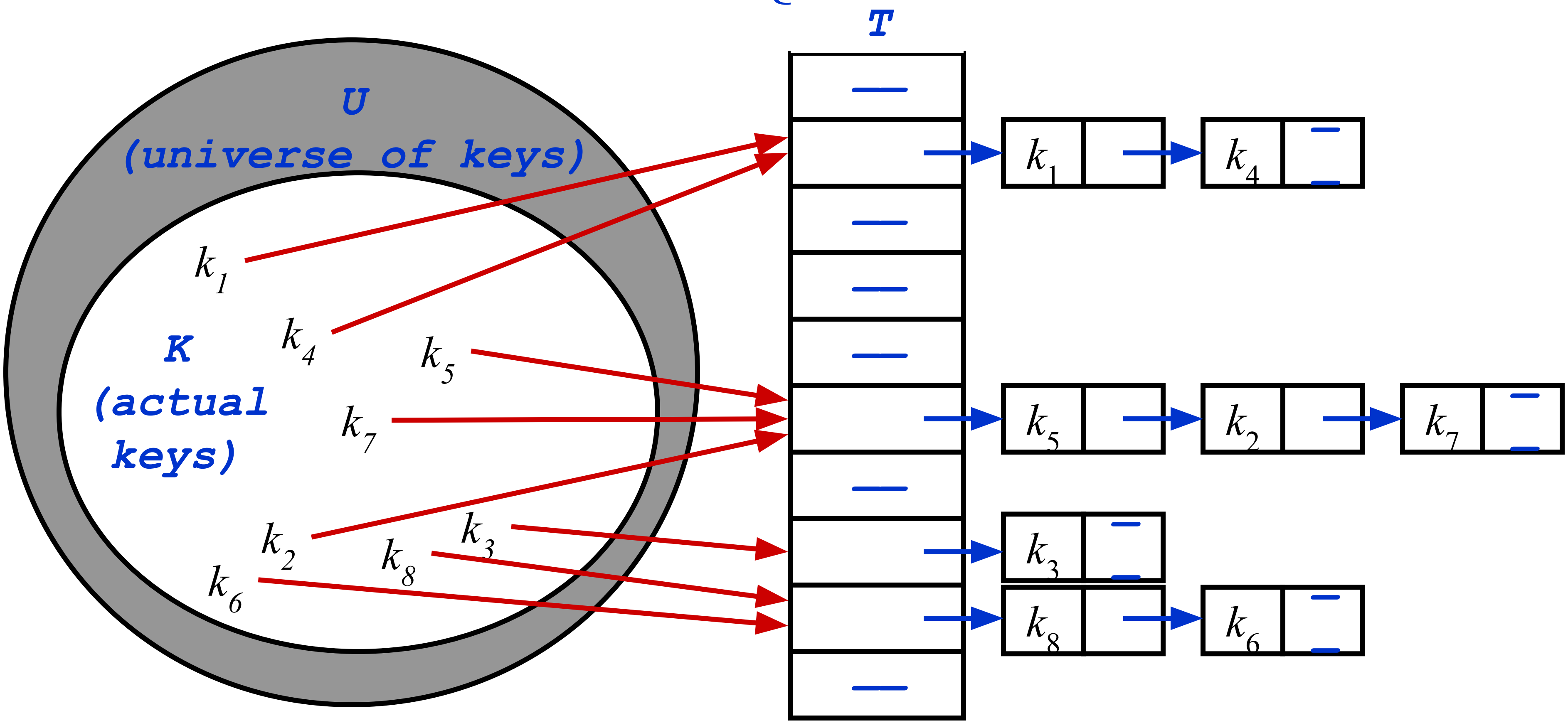
search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

delete x from the list $T[h(key[x])]$

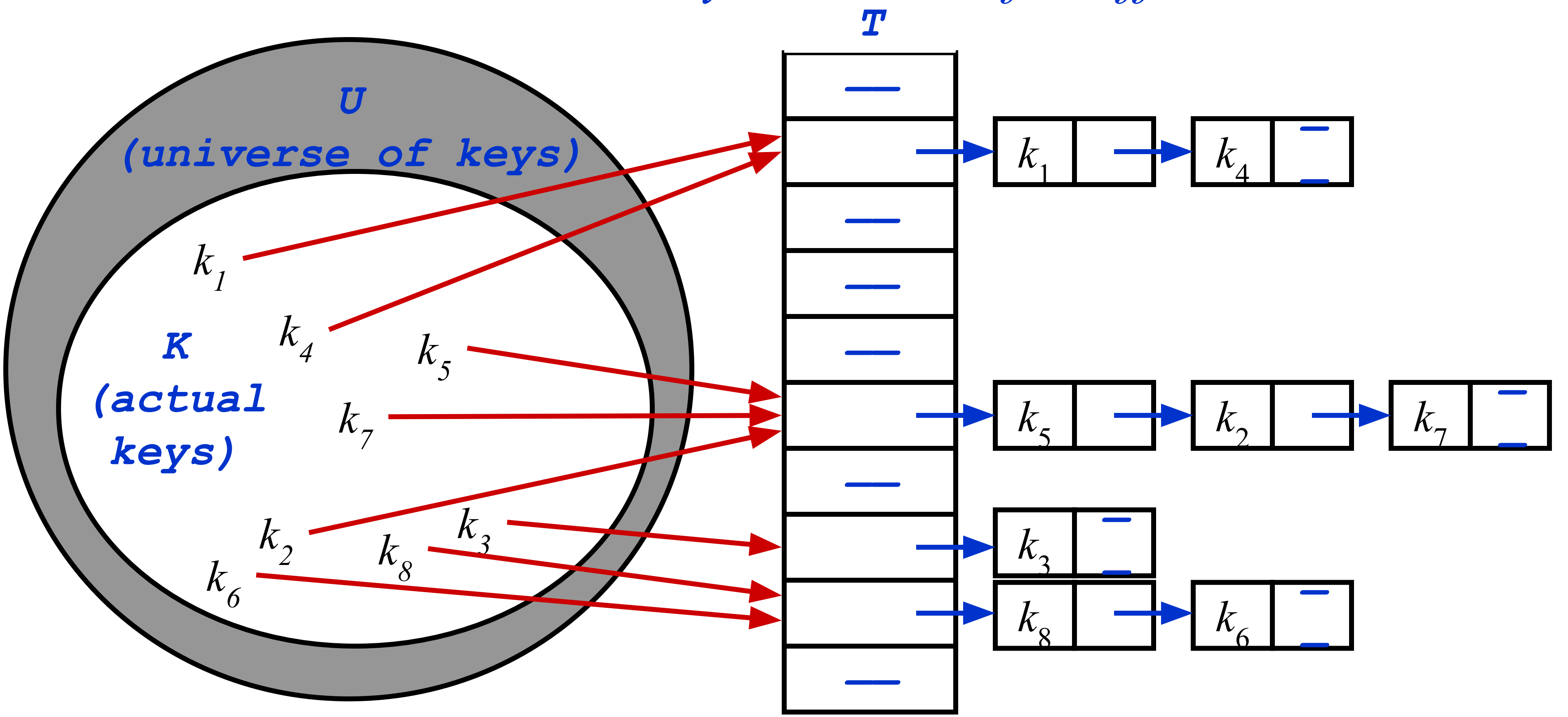
Chaining

- *How do we insert an element?*
  - *The worst-case running time for insertion?*



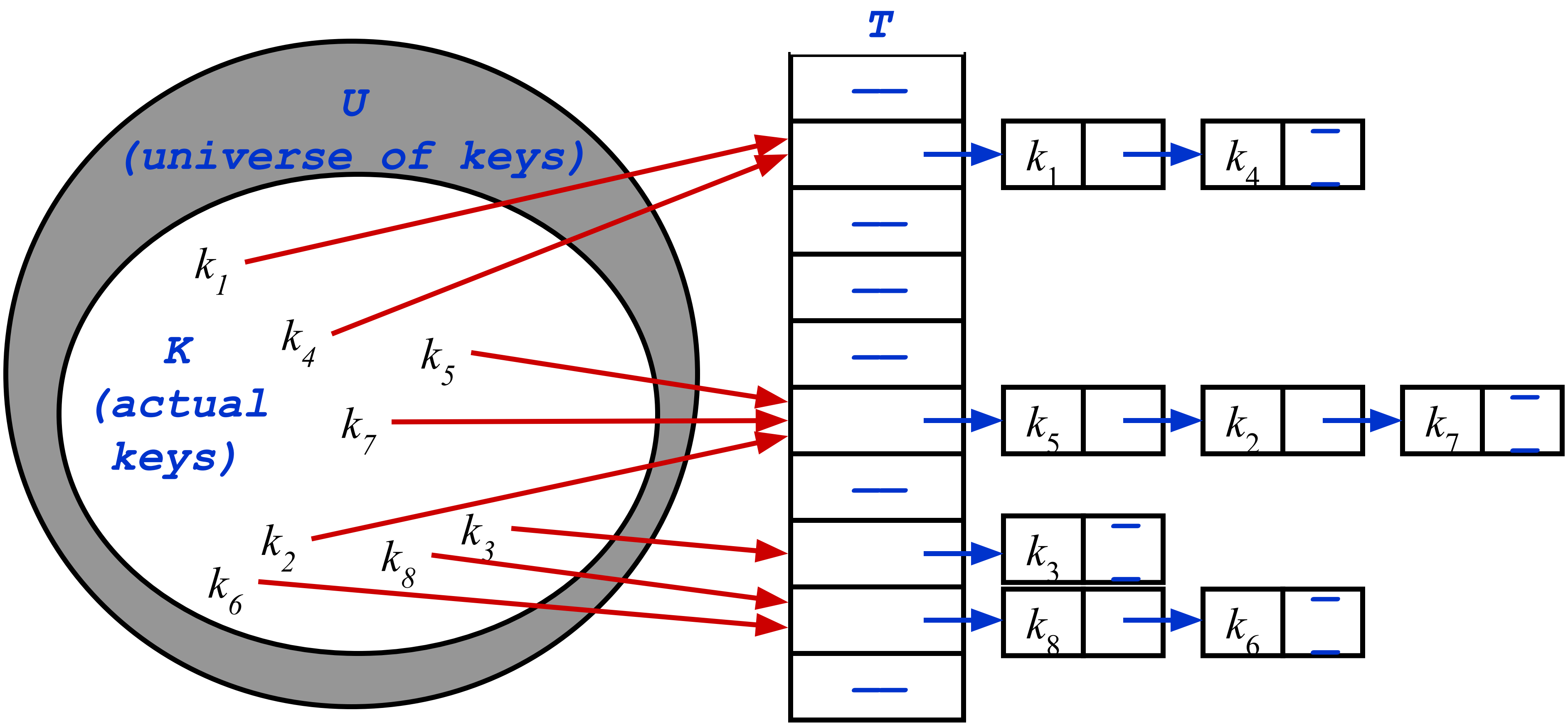
Chaining

- *How do we delete an element?*
  - *Do we need a doubly-linked list for efficient delete?*



Chaining

- *How do we search for a element with a given key?*



Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table: the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the cost of an unsuccessful search for a key?*
  - *Worst case?*
  - *Average case?*
- *What will be the cost of a successful search for a key?*

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$
- *What will be the average cost of a successful search?* A: $O(1 + \alpha/2) = O(1 + \alpha)$

Analysis of Chaining Continued

- So the cost of searching = $O(1 + \alpha)$
- *If the number of keys n is proportional to the number of slots in the table, what is α?*
- A: $\alpha = O(1)$
  - In other words, we can make the expected cost of searching constant if we make α constant

Choosing A Hash Function

- Clearly choosing the hash function well is crucial
  - *What will a worst-case hash function do?*
  - *What will be the time to search in this case?*
- *What are desirable features of the hash function?*
  - Should distribute keys uniformly into slots
  - Should not depend on patterns in the data

Choosing A Hash Function

- 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.

Interpreting keys as natural numbers

- most use the universe of keys is the set $N = \{0, 1, 2, \ldots\}$
- a character string?
- radix notation:
  - pt?
  - the pair of decimal integers (112, 116), since p = 112 and t = 116
  - radix-128 integer, pt becomes $(112 \cdot 128)+116 = 14452$

Hash Functions: The Division Method

- $h(k) = k \bmod m$
  - In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
  - Hashing is quite fast!
- *What happens to elements with adjacent values of k?*
- *What happens if m is a power of 2 (say 2^P)?*
  - *$h(k)$ is just the p lowest-order bits of k*
  - *It is better to make the hash function depend on all the bits of the key.*
- *What if m is a power of 10?*
- Upshot: pick table size m = prime number not too close to a power of 2 (or 10)

Hash Functions: The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m\,(kA - \lfloor kA \rfloor) \rfloor$

What does this term represent?

Hash Functions: The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m \underbrace{(kA - \lfloor kA \rfloor)}_{\textit{Fractional part of } kA} \rfloor$
- the value of m is not critical
- Choose $m = 2^P$
- Choose A not too close to 0 or 1
- Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Hash Functions: Malicious adversary

- Scenario:
  - You are given an assignment to implement hashing
  - You will self-grade in pairs, testing and grading your partner's implementation
  - In a blatant violation of the honor code, your partner:
    - Analyzes your hash function
    - Picks a sequence of "worst-case" keys, causing your implementation to take O(n) time to search

Hash Functions: Universal Hashing

- As before, when attempting to foil an malicious adversary: randomize the algorithm
- *Universal hashing*: pick a hash function randomly in a way that is independent of the keys that are actually going to be stored
  - Guarantees good performance on average, no matter what keys adversary chooses

Review: The Division Method

- $h(k) = k \bmod m$
  - In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
- Elements with adjacent keys hashed to different slots: good
- If keys bear relation to m: bad
- Upshot: pick table size m = prime number not too close to a power of 2 (or 10)

Review: The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m \underbrace{(kA - \lfloor kA \rfloor)}_{\textit{Fractional part of } kA} \rfloor$
- Upshot:
  - Choose $m = 2^P$
  - Choose A not too close to 0 or 1
  - Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Review: Universal Hashing

- When attempting to foil an malicious adversary, randomize the algorithm
- *Universal hashing*: pick a hash function randomly when the algorithm begins (*not* upon every insert!)
  - Guarantees good performance on average, no matter what keys adversary chooses
  - Need a family of hash functions to choose from

Universal Hashing

- Let $\mathcal{H}$ be a (finite) collection of hash functions
  - …that map a given universe U of keys…
  - …into the range $\{0, 1, \ldots, m - 1\}$.
- $\mathcal{H}$ is said to be *universal* if:
  - for each pair of distinct keys $x, y \in U$, the number of hash functions h $\in \mathcal{H}$ for which $h(x) = h(y)$ is $|\mathcal{H}|/m$
  - In other words:
    - With a random hash function from $\mathcal{H}$, the chance of a collision between x and y is exactly $1/m$ $(x \neq y)$

Universal Hashing

- Theorem 12.3:
  - Choose h from a universal family of hash functions
  - Hash n keys into a table of m slots, $n \le m$
  - Then the expected number of collisions involving a particular key x is less than 1
  - Proof:
    - For each pair of keys y, z, let $c_{yx} = 1$ if y and z collide, 0 otherwise
    - $E[c_{yz}] = 1/m$ (by definition)
    - Let C_x be total number of collisions involving key x
    - $$E[C_x] = \sum_{\substack{y \in T \\ y \neq x}} E[c_{xy}] = \frac{n-1}{m}$$
    - Since $n \le m$, we have $E[C_x] < 1$

A Universal Hash Function

- Choose table size m to be prime
- Decompose key x into r+1 bytes, so that $x = \{x_0, x_1, \ldots, x_r\}$
  - Only requirement is that max value of byte < m
  - Let $a = \{a_0, a_1, \ldots, a_r\}$ denote a sequence of r+1 elements chosen randomly from $\{0, 1, \ldots, m - 1\}$
  - Define corresponding hash function $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

  - With this definition, $\mathcal{H}$ has m^{r+1} members

A Universal Hash Function

- $\mathcal{H}$ is a universal collection of hash functions (Theorem 12.4)
- How to use:
  - Pick r based on m and the range of keys in U
  - Pick a hash function by (randomly) picking the a's
  - Use that hash function on all keys

Open Addressing

- Basic idea:
  - To insert: if slot is full, try another slot, …, until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking
- Table needn't be much bigger than n
- *Advantage: no space use for pointers*

Open Addressing

- Insert

```
HASH-INSERT(T, k)
i ← 0
repeat  j ← h(k, i)
        if T[j] = NIL
          then T[j] ← k
               return j
          else  i ← i + 1
  until i = m
error "hash table overflow"
```

Open Addressing

- Insert

```
HASH-INSERT(T, k)
i ← 0
repeat j ← h(k, i)
       if T[j] = NIL
          then T[j] ← k
               return j
          else i ← i + 1
  until i = m
error "hash table overflow"
```

Open Addressing

- Search

```
HASH-SEARCH(T, k)
i ← 0
repeat j ← h(k, i)
       if T[j] = k
          then return j
       i ← i + 1
  until T[j] = NIL or i = m
return NIL
```

Open Addressing

- Deletion from an open-address hash table is difficult. Why?
- We cannot simply mark that slot as empty by storing *null* in it.
- One solution is to mark the slot by storing in it the special value DELETED instead of *null*.
- Modify the procedure HASH-INSERT
- No modification of HASH-SEARCH

Open Addressing

- Three techniques are commonly used to compute the probe sequences required
- linear probing,
- quadratic probing,
- double hashing.

Open Addressing, Linear Probing

$$h(k, i) = (h'(k) + i) \bmod m$$

- Given key k, the first slot probed is T [h'(k)]
- We next probe slot T [h'(k) + 1], and
- so on up to slot T [m − 1].
- Then we wrap around to slots T [0], T [1], . . ., until we finally probe slot T [h(k) − 1].
- Suffers from a problem known as primary clustering: Long runs of occupied slots build up, increasing the average search time.

Open Addressing, Quadratic Probing

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \ ,$$

- This property leads to a milder form of clustering, called secondary clustering.

Open Addressing, Double Hashing

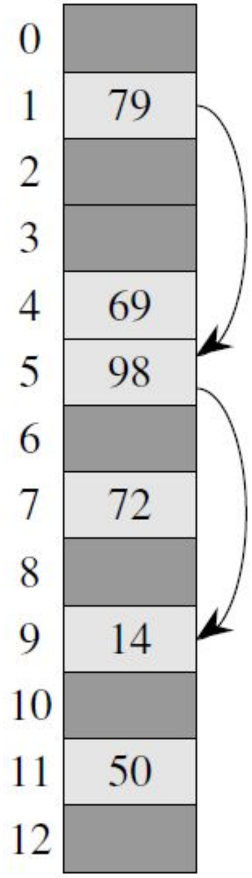
$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m \ ,$$

- The initial probe is to position T [h1(k)]
- Successive probe positions are offset from previous positions by the amount h2(k), modulo m
- The probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary.

Open Addressing, 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.

79, 69, 72, 98, 50, 14
m: 13



$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m ,$$

The End