Lecture 11 Hash Tables

Sultan ALPAR
associate professor, IITU
s.alpar@iitu.edu.kz

The Search Problem

- Unsorted list
  - O(N)
- Sorted list
  - O(logN) using arrays (i.e., binary search)
  - O(N) using linked lists
- Binary Search tree
  - O(logN) (i.e., balanced tree)
  - O(N) (i.e., unbalanced tree)
- Can we do better than this?
  - Direct Addressing
  - Hashing

Direct Addressing

- Assumptions:
  - Key values are **distinct**
  - Each key is drawn from a universe $U = \{0, 1, \ldots, n - 1\}$
- Idea:
  - Store the items in an array, indexed by keys

Direct Addressing (cont'd)

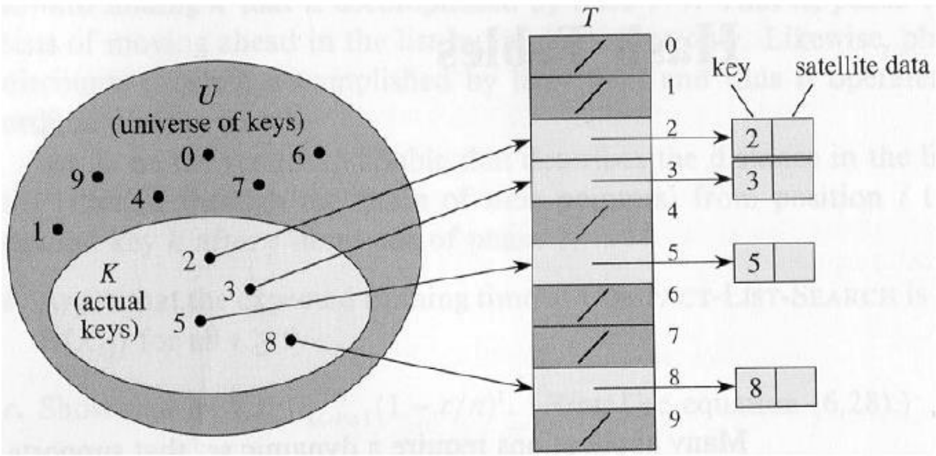
- **Direct-address table** representation:

  – An array T[0 . . . n - 1]

  – Each **slot**, or position, in T corresponds to a key in U

  – For an element x with key k, a pointer to x will be placed in location T[k]

  – If there are no elements with key k in the set, T[k] is empty, represented by NIL



Search, insert, delete in O(1) time!

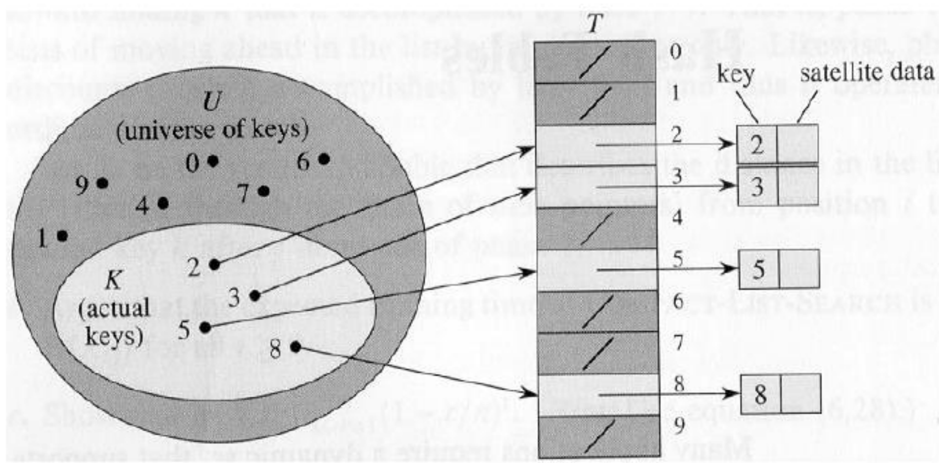
Direct Addressing (cont'd)

<u>Example 1:</u> Suppose that the are integers from 1 to 100 and that there are about 100 records.

Create an array A of 100 items and stored the record whose key is equal to **i** in in **A[i].**

|K| = |U|

|K|: # elements in K
|U|: # elements in U

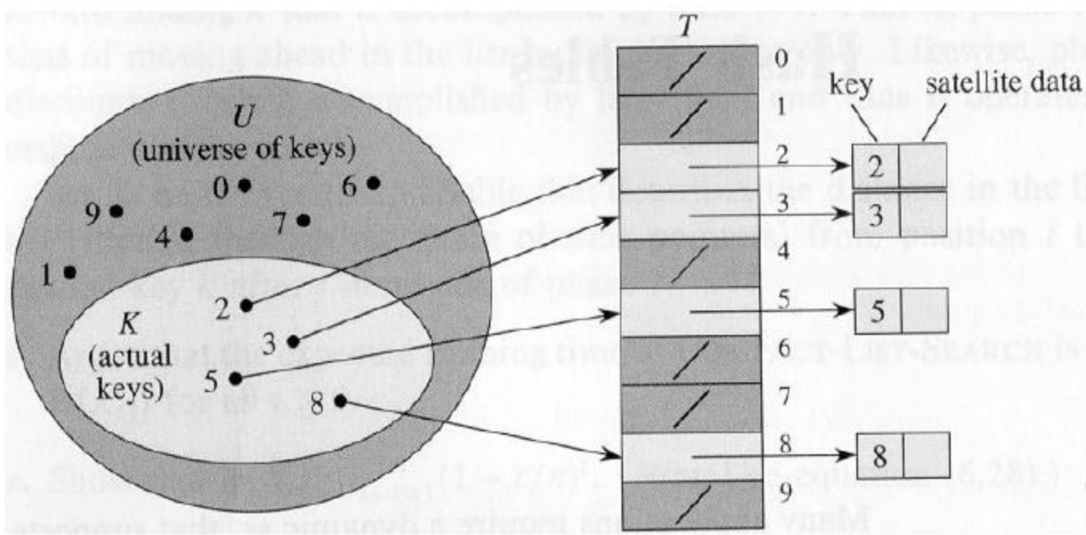


Direct Addressing (cont'd)

<u>Example 2:</u> Suppose that the keys are 9-digit social security numbers (SSN)

Although we could use the same idea, it would be very inefficient (i.e., use an array of 1 billion size to store 100 records)

|K| << |U|



Hashing

Idea:

– Use a function **h** to compute the slot for each key

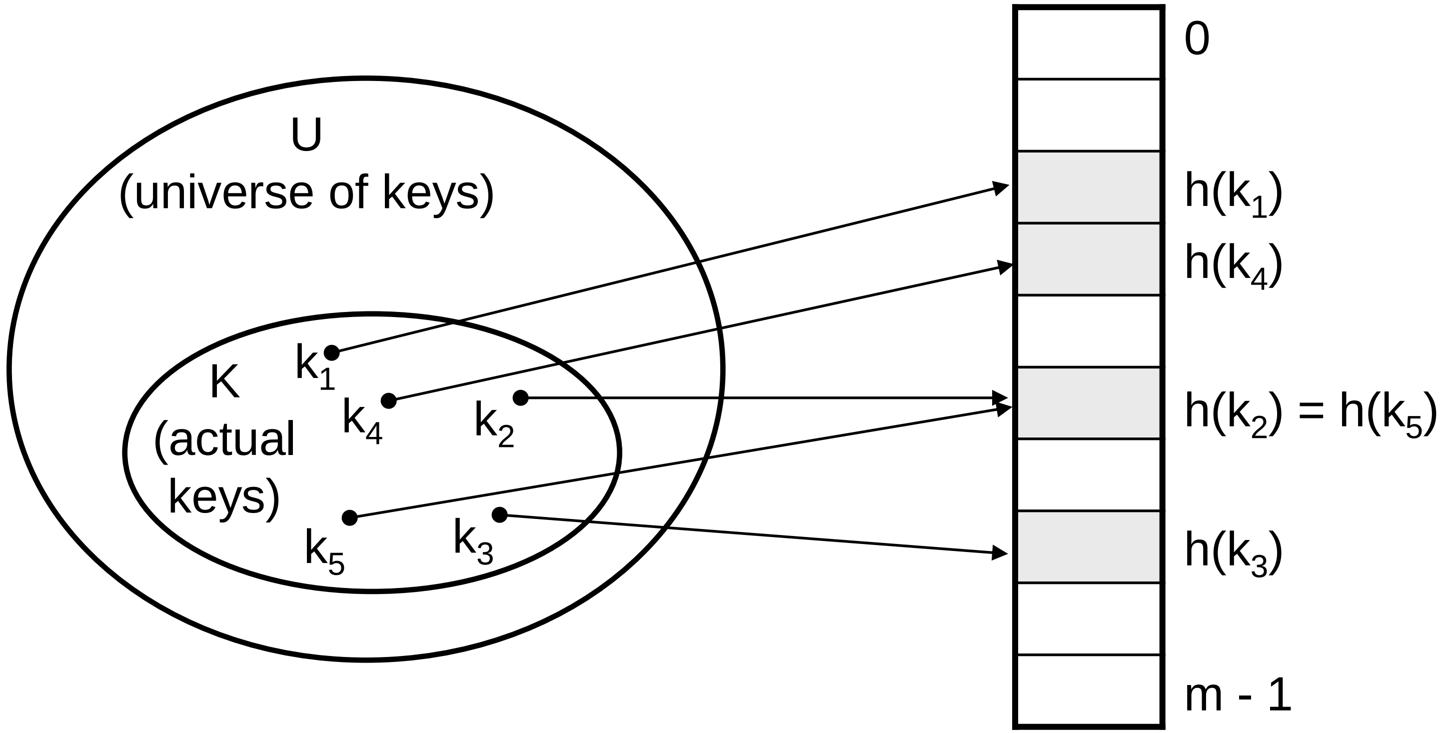
– Store the element in slot **h(k)**

- A **hash function** h transforms a key into an index in a hash table $T[0\dots m-1]$:

$$\boxed{h : U \rightarrow \{0, 1, \dots, m - 1\}}$$

- We say that k **hashes** to slot $h(k)$

Hashing (cont'd)



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

hash table size: **m**

Hashing (cont'd)

<u>Example 2:</u> Suppose that the keys are 9-digit social security numbers (SSN)

Possible hash function

$$h(ssn) = sss \text{ mod } 100 \text{ (last 2 digits of ssn)}$$

e.g., if $ssn = 10123411$ then $h(10123411) = 11$)

Advantages of Hashing

- Reduce the range of array indices handled:

  m instead of $|U|$

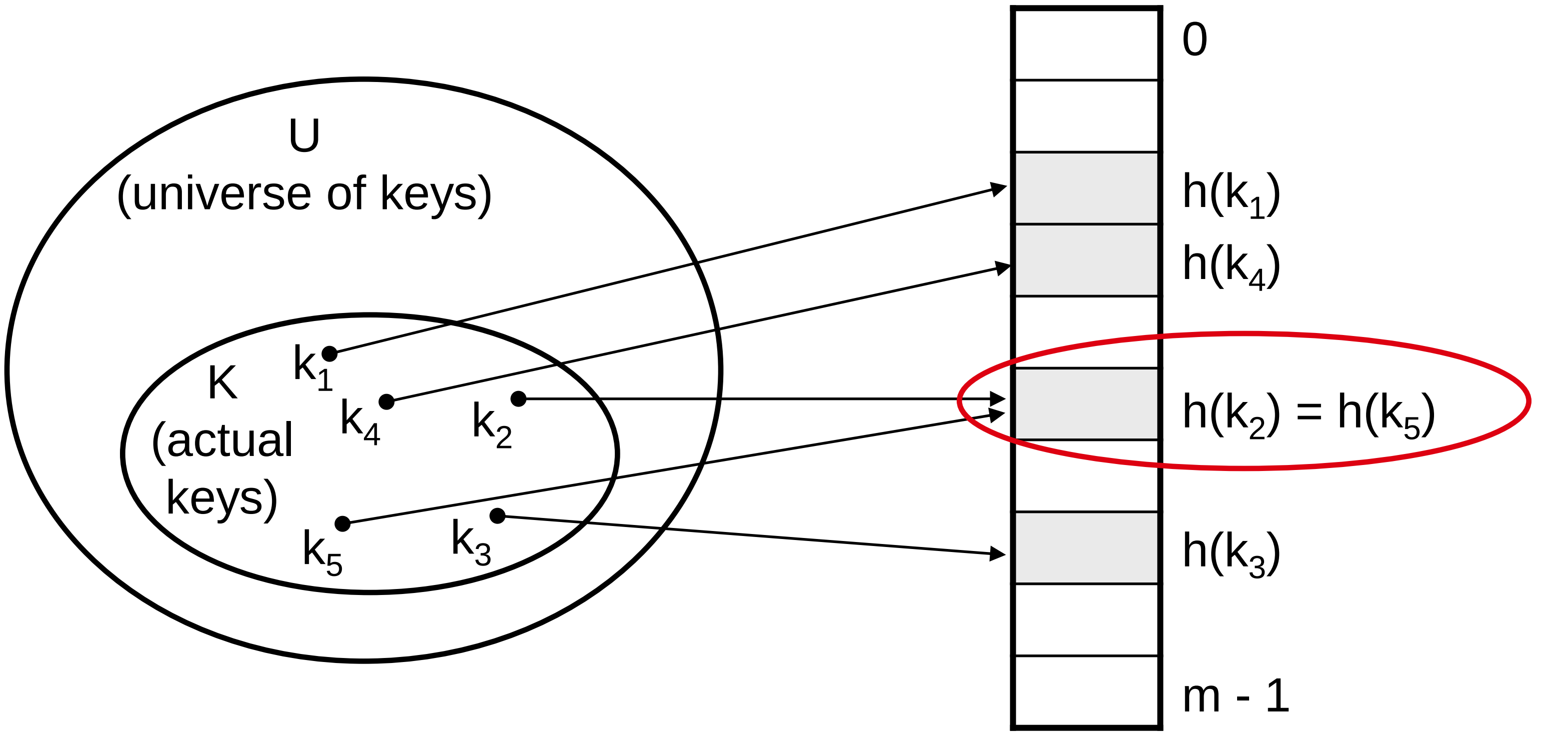
  where m is the hash table size: T[0, …, m-1]

- Storage is reduced.

- $O(1)$ search time (i.e., under assumptions).

Collisions

Collisions occur when $h(k_i)=h(k_j)$, $i \neq j$



Collisions (cont'd)

- For a given set K of keys:
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function!
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely might not be easy.

Handling Collisions

- We will discuss two main methods:

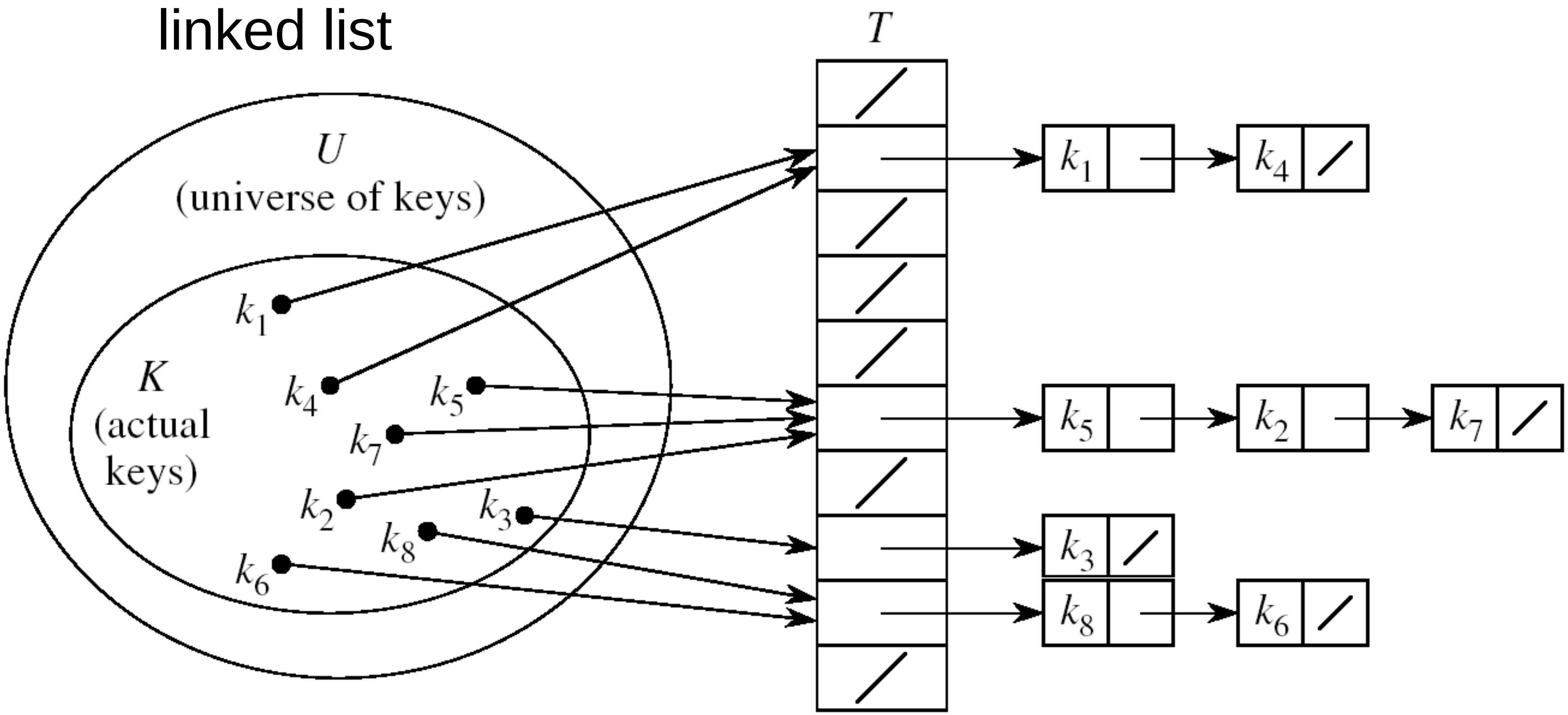
(1) Chaining

(2) Open addressing

- Linear probing
- Quadratic probing
- Double hashing

Chaining

- **Idea**:
  - Put all elements that hash to the same slot into a linked list



  - Slot j contains a pointer to the head of the list of all elements that hash to j

Chaining (cont'd)

- How to choose the size of the hash table **m**?
  - Small enough to avoid wasting space.
  - Large enough to avoid many collisions and keep linked-lists short.
  - Typically 1/5 or 1/10 of the total number of elements.
- Should we use sorted or unsorted linked lists?
  - Unsorted
    - Insert is fast
    - Can easily remove the most recently inserted elements

Hash Table Operations

- Search
- Insert
- Delete

Searching in Hash Tables

Alg.: CHAINED-HASH-SEARCH(T, k)

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

- Running time <u>depends</u> on the length of the list of elements in slot $h(k)$

Insertion in Hash Tables

Alg.: CHAINED-HASH-INSERT(T, x)

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

- T[h(key[x])] takes O(1) time; insert will take O(1) time overall since lists are unsorted.

- Note: if no duplicates are allowed, It would take extra time to check if item was already inserted.

Deletion in Hash Tables

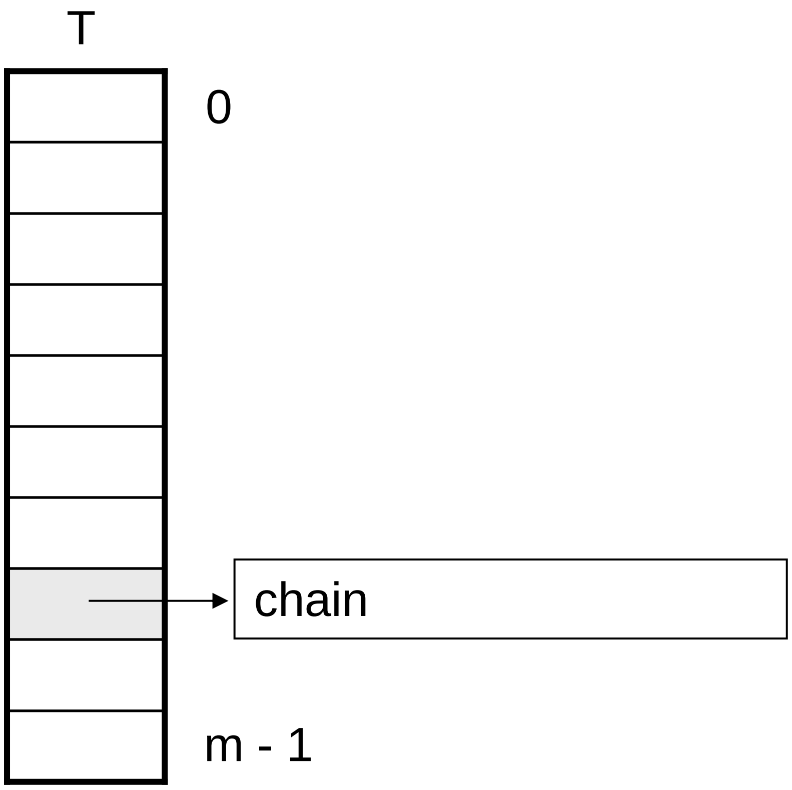
Alg.: CHAINED-HASH-DELETE(T, x)

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

- T[h(key[x])] takes O(1) time.
- Finding the item <u>depends</u> on the length of the list of elements in slot h(key[x])

Analysis of Hashing with Chaining: Worst Case

- How long does it take to search for an element with a given key?



- Worst case:
  - All n keys hash to the same slot

then $O(n)$ plus time to compute the hash function

Analysis of Hashing with Chaining: Average Case

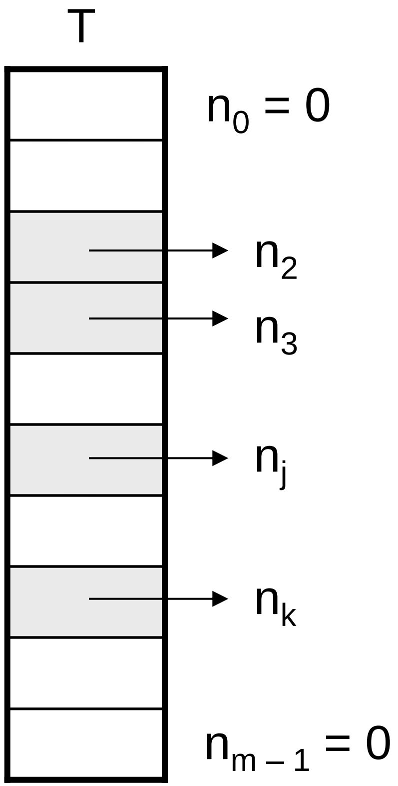
- It depends on how well the hash function distributes the n keys among the m slots

- Under the following assumptions:

  (1) **n = O(m)**

  (2) any given element is **equally likely** to hash into any of the m slots (i.e., simple uniform hashing property)

then → O(1) time plus time to compute the hash function



Properties of Good Hash Functions

- **Good hash function properties**

  (1) Easy to compute

  (2) Approximates a random function

  i.e., for every input, every output is equally likely.

  (3) Minimizes the chance that similar keys hash to the same slot

  i.e., strings such as pt and pts should hash to different slot.

- **We will discuss two methods:**
  - Division method
  - Multiplication method

The Division Method

- **Idea:**
  - Map a key k into one of the m slots by taking the remainder of k divided by m

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

- **Advantage**:
  - fast, requires only one operation
- **Disadvantage**:
  - Certain values of m are bad (i.e., collisions), e.g.,
    - power of 2
    - non-prime numbers

Example

- If $m = 2^p$, then $h(k)$ is just the least significant p bits of k
  - $p = 1 \Rightarrow m = 2$

    $\Rightarrow h(k) = \{0, 1\}$, least significant 1 bit of k
  - $p = 2 \Rightarrow m = 4$

    $\Rightarrow h(k) = \{0, 1, 2, 3\}$, least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2
  - Column 2: k mod 97
  - Column 3: k mod 100

	m 97	m 100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

The Multiplication Method

Idea:

(1) Multiply key k by a constant A, where $0 < A < 1$

(2) Extract the fractional part of kA

(3) Multiply the fractional part by m

(4) Truncate the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m\, \underbrace{(k\, A \bmod 1)}_{\text{fractional part of kA = kA - } \lfloor kA \rfloor} \rfloor$$

$$e.g., \lfloor 12.3 \rfloor = 12$$

- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical

Example – Multiplication Method

Suppose k=6, A=0.3, m=32

(1) k x A = 1.8

(2) fractional part: $1.8 - \lfloor 1.8 \rfloor = 0.8$

(3) m x 0.8 = 32 x 0.8 = 25.6

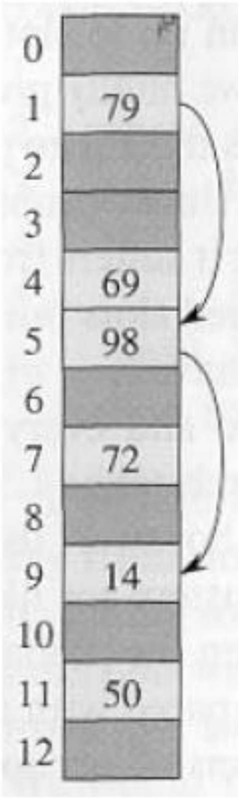
(4) $\lfloor 25.6 \rfloor = 25$

h(6)=25

Open Addressing

- Idea: store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - <u>Insertion:</u> if a slot is full, try another one, until you find an empty one.
  - <u>Search:</u> follow the same probe sequence.
  - <u>Deletion:</u> need to be careful!
- Search time depends on the length of probe sequences!

e.g., insert 14



probe sequence: <1, 5, 9>

Generalize hash function notation:

- A hash function contains two arguments now:

  (i) key value, and (ii) probe number

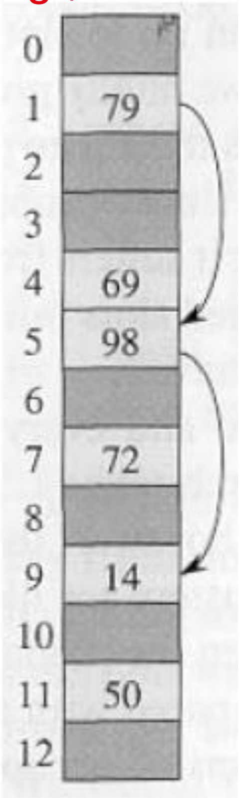
  $h(k,p), \quad p=0,1,\ldots,m-1$

- Probe sequence:

  $<h(k,0), h(k,1), h(k,2), \ldots >$

- Example:

e.g., insert 14



Probe sequence: $<h(14,0), h(14,1), h(14,2)>=<1, 5, 9>$

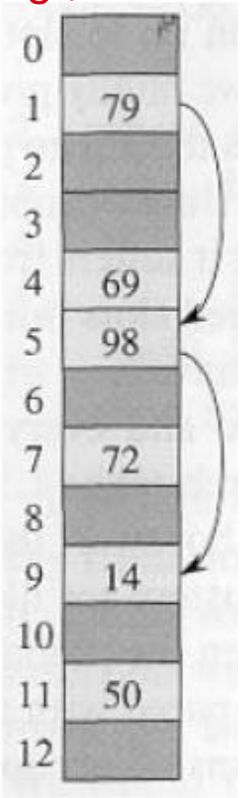
Generalize hash function notation:

– Probe sequence must be a permutation of

<0,1,...,m-1>

– There are m! possible permutations

e.g., insert 14



Probe sequence: <h(14,0), h(14,1), h(14,2)>=<1, 5, 9>

Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

- None of these methods can generate more than m^2 different probe sequences!

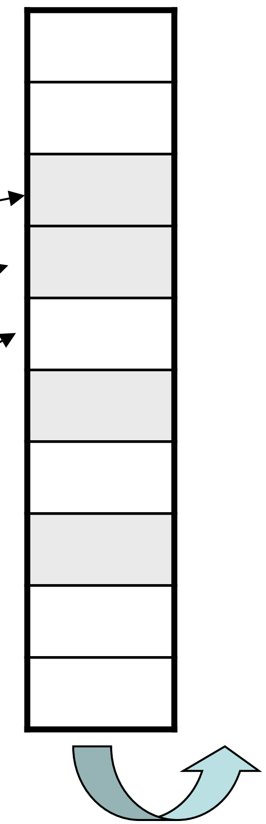
Linear probing: Inserting a key

- **Idea:** when there is a collision, check the <u>next</u> available position in the table:

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

$$i=0,1,2,...$$

- $i=0$: first slot probed: $h_1(k)$
- $i=1$: second slot probed: $h_1(k) + 1$
- $i=2$: third slot probed: $h_1(k)+2$, and so on



probe sequence: $< h_1(k), h_1(k)+1 , h_1(k)+2 , ... >$

- How many probe sequences can linear probing generate?

m probe sequences maximum

wrap around

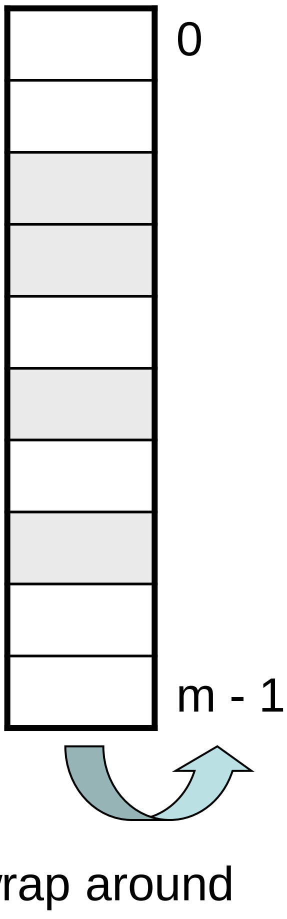
Linear probing: Searching for a key

- Given a key, generate a probe sequence using the same procedure.
- Three cases:

(1) Position in table is occupied with an element of equal key→ FOUND

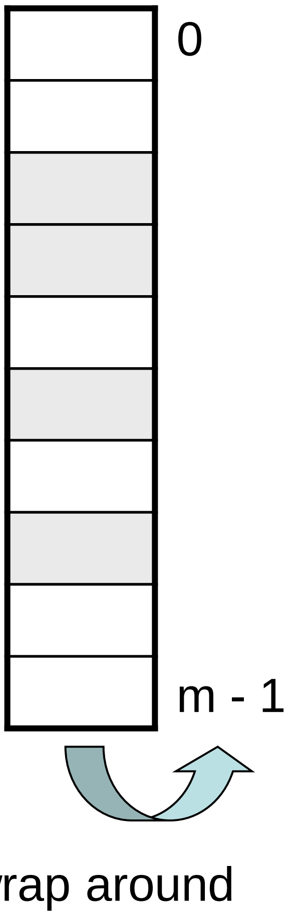
(2) Position in table occupied with a different element → KEEP SEARCHING

(3) Position in table is empty→ NOT FOUND



Linear probing: Searching for a key

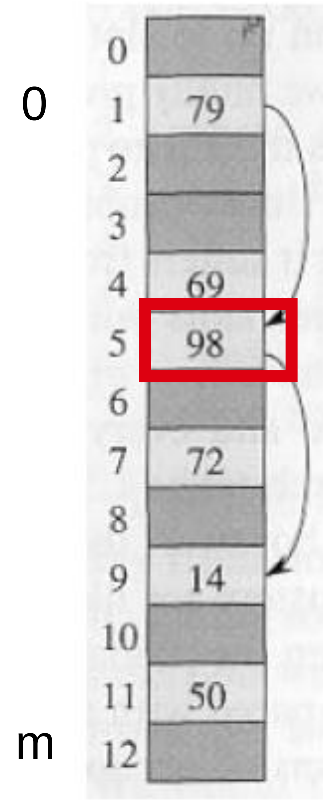
- Running time depends on the length of the probe sequences.
- Need to keep probe sequences short to ensure fast search.



Linear probing: Deleting a key

- First, find the slot containing the key to be deleted.
- Can we just mark the slot as empty?
  - It would be impossible to retrieve keys inserted after that slot was occupied!
- Solution
  - "Mark" the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion.

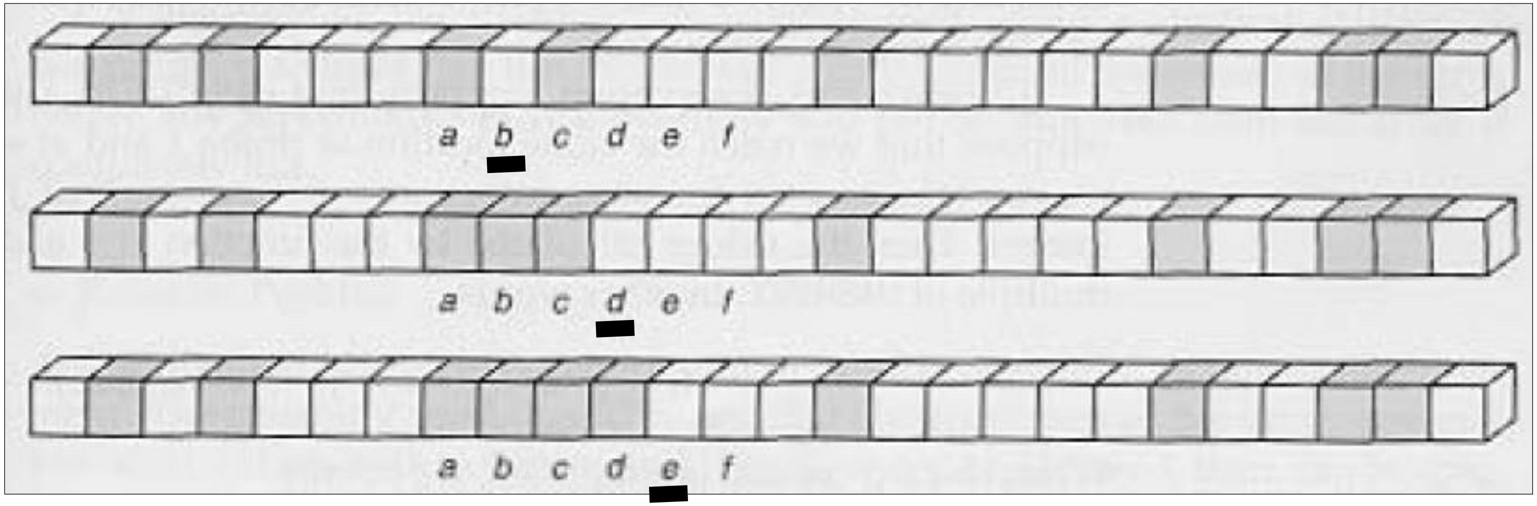
e.g., delete 98



Primary Clustering Problem

- Long chunks of occupied slots are created.
- As a result, some slots become more likely than others.
- Probe sequences increase in length. ⇒ search time increases!!

initially, all slots have probability 1/m



Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Quadratic probing

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

i=0,1,2,...

- Clustering is less serious but still a problem *(secondary clustering)*
- How many probe sequences can quadratic probing generate?

m -- the initial position determines probe sequence

Double Hashing

(1) Use one hash function to determine the first slot.

(2) Use a second hash function to determine the increment for the probe sequence:

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m, \quad i=0,1,\ldots$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on ...
- Advantage: handles clustering better
- Disadvantage: more time consuming
- How many probe sequences can double hashing generate?

m^2 -- why?

Double Hashing: Example

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- Insert key 14:

i=0: $h(14,0) = h_1(14) = 14 \bmod 13 = 1$

i=1: $h(14,1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

i=2: $h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) \bmod 13 = 9$