Hashing

Lecture 21

Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

Adapted partially from Data Structures and Algorithms in Java, M.T. Goodrich, R.Tamassia and M. H. Goldwasser, Sixth Edition, Wiley; Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning

Introduction

- Main operations used by searching??
  - comparing keys
- e.g., In *sequential* search
  - search the table (or array) storing the elements *in order*
  - key comparison determines a match
- e.g., In *binary* search
  - the table (or array) storing the elements is *divided into halves*
  - determine which half to check
  - key comparison determines a match
- A different way to search??
  - calculate the ***position*** of the **key** in the table based on the ***value of key***

Introduction

- A different way to search?
  - calculate the ***position*** of the ***key*** in the table based on the ***value of key***
  - the value of key ***indicates*** the ***position***
    - when the ***key*** is known, the ***position*** can be accessed directly
      - no other preliminary tests
      - search time: O(1)
        - regardless of the number of elements being searched
        - the run time is ***same***

Introduction (cont.)

- For example
  - a small company of 100 employees, assigned an ***employ id*** in the range of 0 – 99
  - ***employ id*** → ***index*** into the array (or table)
  - directly access the record of any employee, if ***employ id*** is known

Key	Array of Employees' Records
Key 0 —mapping→ [0]	Employee record with Emp_ID 0
Key 1 —mapping→ [1]	Employee record with Emp_ID 1
Key 2 —mapping→ [2]	Employee record with Emp_ID 2
....	....
mapping	
....	....
Key 98 → [98]	Employee record with Emp_ID 98
Key 99 → [99]	Employee record with Emp_ID 99

mapping ←

2-digit key

Employee Table

100 entries

Introduction (cont.)

- For example (cont.)
  - what if, five-digit ***employ id*** used as the primary key?
  - key value ranging from 00000 to 99999 → 100,000 array size

Key	Array of Employees' Records
Key 00000 ⟶ [0]	Employee record with Emp_ID 00000
....	....
Key n ⟶ [n]	Employee record with Emp_ID n
....	....
Key 99998 ⟶ [99998]	Employee record with Emp_ID 99998
Key 99999 ⟶ [99999]	Employee record with Emp_ID 99999

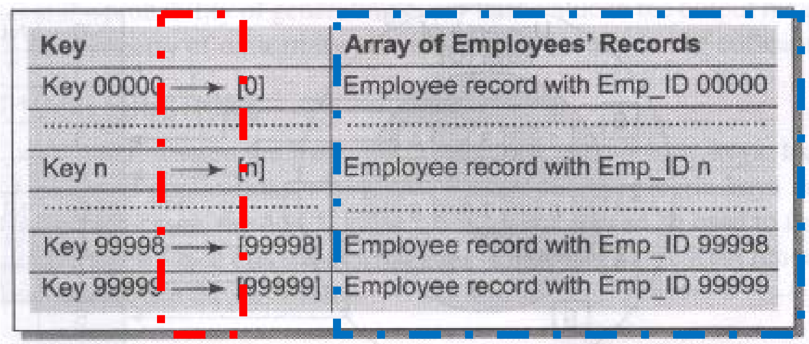
Employee Table
100,000 entries

    - actually there are only 100 employees in the company
  - just use ***last two digits*** of the ***key*** to identify each employee

Introduction (cont.)

- For example (cont.), need to use ***five-digit*** ***employee id***
  - convert a ***five-digit key number*** to a ***two-digit array index***?
    - need a ***specific function***…
    - e.g., Emp_ID 79439 → index 39
    - e.g., Emp_ID 12345 → index 45

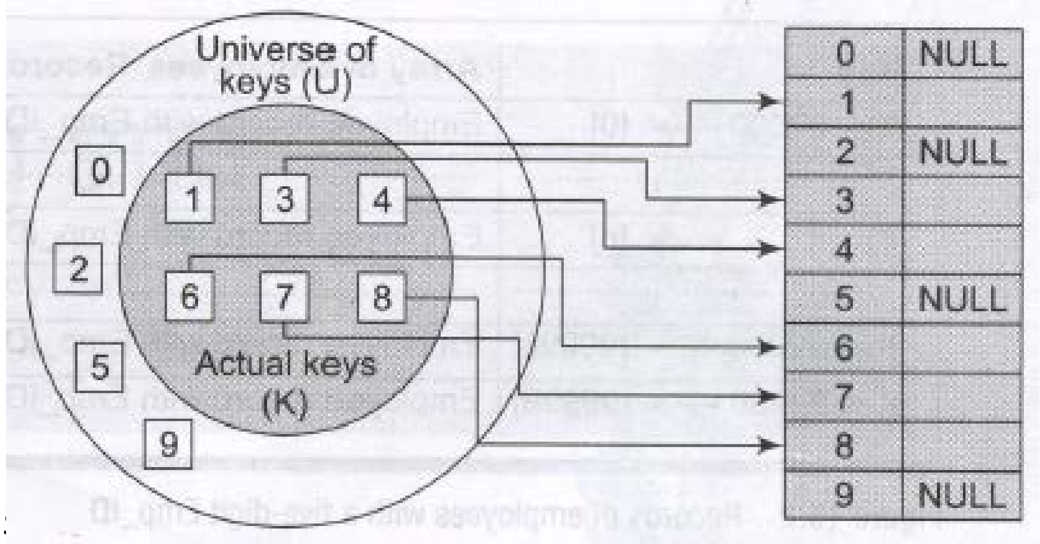
hash function hash table



- Terminologies
  - **hash table** ←→ an array
  - **hash function** ←→ carry out the transformation
- ***Hash function, h***
  - transform a ***key*** (e.g., *string*, *number*, *record*, or *the like*), ***K***, into an ***index*** for a table used to store items of the same type as *K*
- ***Perfect hash function, h***
  - *h* transforms different keys into different indexes **Our goal!**

Hash Tables

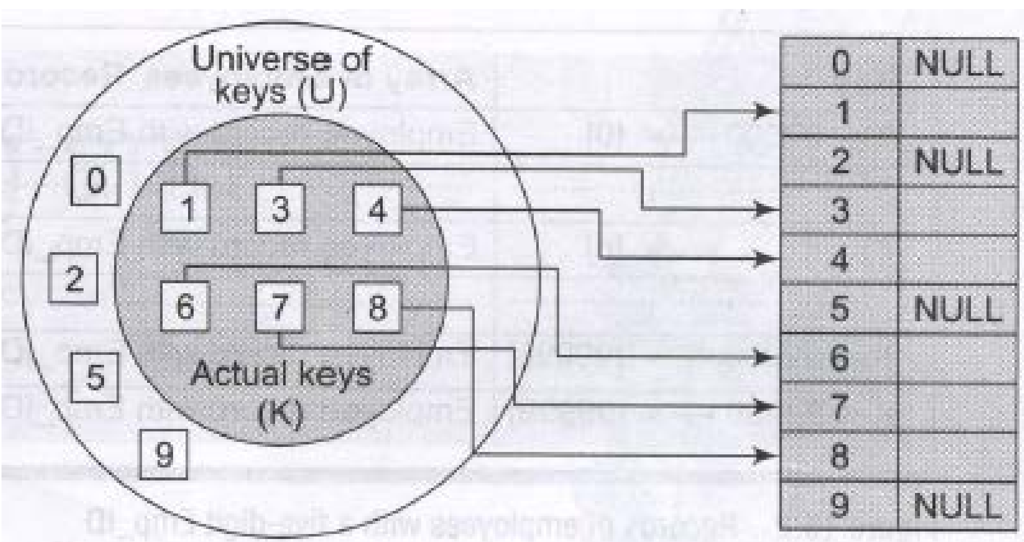
- ***Hash table***: a data structure where
  - ***keys*** are mapped to array positions (***index***) by a **hash function**
- For example, a **direct correspondence** between the keys and the indices of the array
  - useful when the total universe of keys is small
  - useful when most of the keys are used from the whole set of keys



storage requirement for a hash table, O(k), k is the number of keys actually used

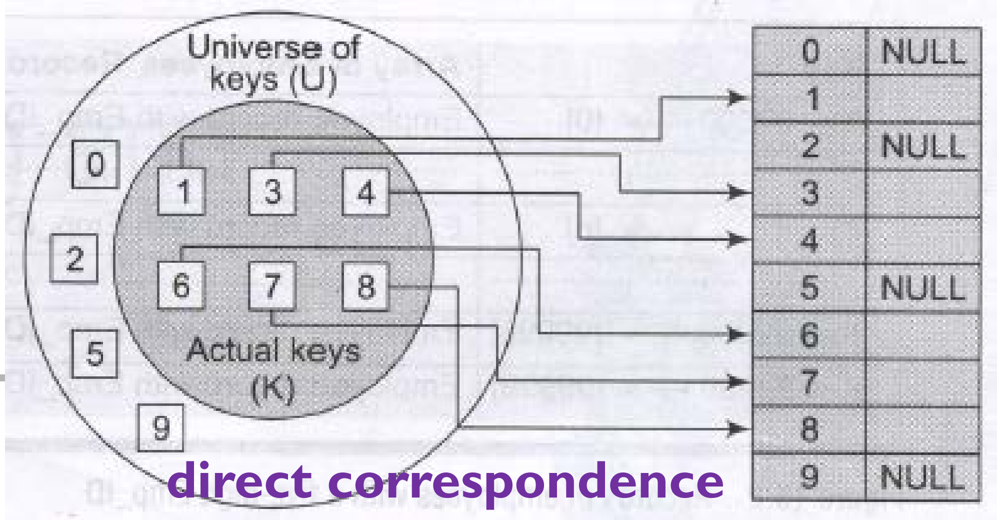
Hash Tables (cont.)

- ***Hashing***
  - process of mapping the **keys** to appropriate **locations (or indices)** in a hash table
  - e.g., an element with **key k** stored at index h(k) , **NOT k**
    - use a **hash function** h



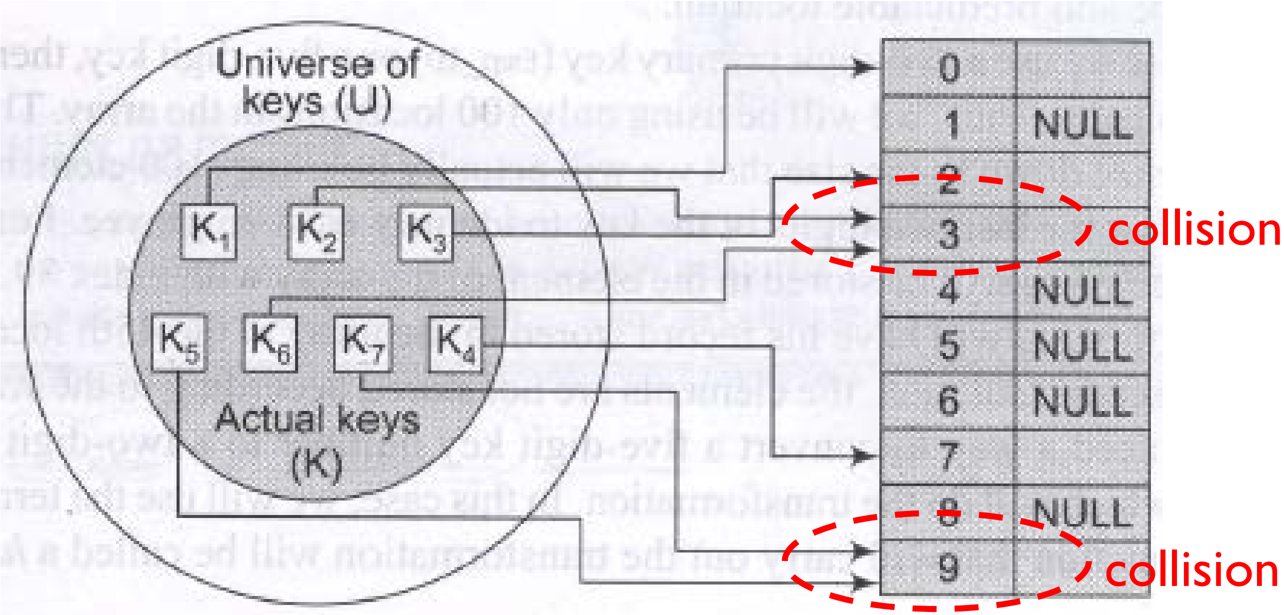
storage requirement for a hash table, O(k), k is the number of keys actually used

Hash Tables (cont.)



direct correspondence

- **Collision**
  - ***two or more keys*** map to the ***same location***
    - e.g., k_2 and k_6 point to the same location
    - e.g., k_5 and k_7 point to the same location



Hash Functions

- A mathematical formula
  - apply to a ***key*** (numeric or alphanumeric (i.e., ASCII)), and
  - produce an integer used as an **index** for the ***key*** in the ***hash table***
  - ***ideally*** produce a *unique* set of *integers* to ***reduce*** the number of **collisions**
- A ***good hash function***?
  - ***minimize*** the number of ***collisions*** by spreading the elements **uniformly** throughout the array
  - **uniformity** – map the keys as ***evenly*** as possible over output range
    - minimize the number of ***collisions***
  - ***low cost*** – the ***cost*** of executing a hash function
  - **determinism** – the ***same hash value*** must be generated for a given ***same input value***

Hash Functions (cont.): Division

- The number that a hash function returns should be a ***valid*** index of the table
- ***Division modulo***
  - *h(x)* = *x mod T_size*,
    - where *T_size* = *sizeof*(table)
  - it is a best choice if *T_size* is a *prime* number
    - e.g., *T_size* = *7* (prime number)
      - remainder: 0, 1, 2, 3, 4, 5, 6
- For example, calculate the hash values of keys 1234 and 5462
  - here, T_size = 97
    - h(1234) = 1234 % 97 = 70
    - h(5642) = 5642 % 97 = 16

Hash Functions (cont.): Folding

- Two steps:
  - **divide** the ***key value*** into a number of parts such as $k_1, k_2, \ldots, k_n$
    - each part is same number of digits except the last part
  - **add** individual parts
    - $k_1 + k_2 + \ldots + k_n$
    - ignore the last carry, if any
- For example, given a hash table of 100 locations, calculate the hash value using folding method for keys 5678 and 34567
  - key: 5678 → parts: 56 and 78 → sum: 134
    - hash value: 34 (ignore the last carry)
  - key: 34567 → parts: 34, 56, and 7 → sum: 97
    - hash value: 97

Hash Functions (cont.): Mid-Square Function

- Two steps:
  - **square** the value of key, k^2
  - **extract** the middle ***r digits*** of the result
  - h(k) = x, where x is obtained by selecting ***r digits*** from k^2
- A good mid-square hash function
  - most or all digits of the key value contribute to the result
  - not dominated by the distribution of the bottom digits or the top digits of the original key value
- For example, given a hash table of 100 locations, calculate the hash value for keys 1234 and 5642
  - 100 memory locations → indices vary from 0 to 99
    - need only two digits to map the key to a location in the hash table, r = 2
  - k = 1234 → k^2 = 1522756 → h(1234) = 27
  - k = 5642 → k^2 = 31832164 → h(5642) = 32

Collisions

- Map ***two different keys*** to the ***same location*** in the ***hash table***
  - ***cannot*** store two records in the same location
  - solve the problem of ***collision*** → ***collision resolution***
  - *cannot guarantee to eliminate collisions*
- Two most popular methods
  - *open addressing*
  - etc.

Collisions (cont.): Open Addressing

- Open addressing (or closed hashing)
  - upon ***collision***, ***compute new positions***

- Two types of values in hash table
  - sentinel values (e.g., -1 → null): no data value in the location
  - data values

- **Probing**
  - process of examining memory locations in the hash table
  - linear probing, etc.

Open Addressing (cont.): Linear Probing

- **h(k, i) = [h'(k) + i] mod m**, where
  - m: size of the hash table
  - h'(k) = (k mod m)
  - i: the probe number varies from 0 to m – 1
- When inserting a key
  - probe the location generated by h'(k) = k mod m
    - if free, store the value
  - if occupied, subsequently probe the locations generated by
    - [h'(k) + *1*] mod m, [h'(k) + **2**] mod m, [h'(k) + ***3***] mod m, and so on

second probe third probe fourth probe

Open Addressing (cont.): Linear Probing (cont.)

- For example, consider a hash table of size =10.
  - linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

0	1	2	3	4	5	6	7	8	9
−1	81	72	63	24	92	36	27	101	−1

  - 72
    - h(72, 0) = [h'(72) + 0] mod 10 = [72 mod 10 + 0] mod 10 = 2
  - 27
    - h(27, 0) = [h'(27) + 0] mod 10 = [27 mod 10 + 0] mod 10 = 7
  - 36
    - h(36, 0) = [h'(36) + 0] mod 10 = [36 mod 10 + 0] mod 10 = 6

Open Addressing (cont.): Linear Probing (cont.)

- For example, consider a hash table of size =10.
  - linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

0	1	2	3	4	5	6	7	8	9
−1	81	72	63	24	92	36	27	101	−1

  - 24
    - h(24, 0) = [h'(24) + 0] mod 10 = [24 mod 10 + 0] mod 10 = 4
  - 63
    - h(63, 0) = [h'(63) + 0] mod 10 = [63 mod 10 + 0] mod 10 = 3
  - 81
    - h(81, 0) = [h'(81) + 0] mod 10 = [81 mod 10 + 0] mod 10 = 1

Open Addressing (cont.): Linear Probing (cont.)

- For example, consider a hash table of size =10.
  - linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	101	–1

  - 92
    - h(92, 0) = [h'(92) + 0] mod 10 = [92 mod 10 + 0] mod 10 = 2
    - h(92, 1) = [h'(92) + 1] mod 10 = [92 mod 10 + 1] mod 10 = 3
    - h(92, 2) = [h'(92) + 2] mod 10 = [92 mod 10 + 2] mod 10 = 4
    - h(92, 3) = [h'(92) + 3] mod 10 = [92 mod 10 + 3] mod 10 = **5**

Open Addressing (cont.): Linear Probing (cont.)

- For example, consider a hash table of size =10.
  - linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

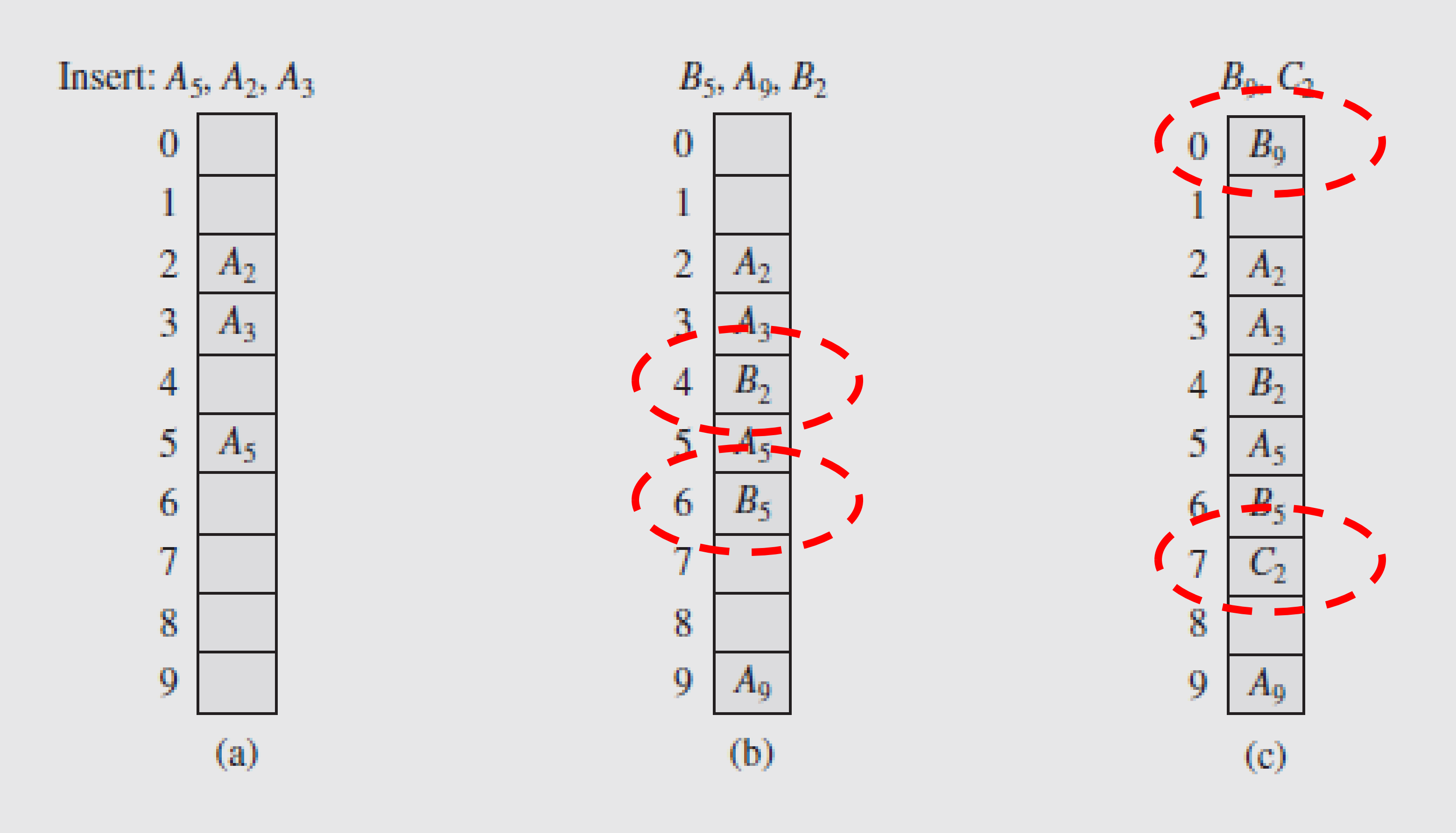
0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	101	–1

sentinel values

  - 101
    - h(101, 0) = [h'(101) + 0] mod 10 = [101 mod 10 + 0] mod 10 = 1
    - …
    - h(101, 7) = [h'(101) + 7] mod 10 = [101 mod 10 + 7] mod 10 = **8**

Open Addressing (cont.): Linear Probing (cont.)

- Another example



Open Addressing (cont.): Linear Probing (cont.)

- **Searching** a value using ***linear probing***
  - **re-compute** the array ***index***
  - **compare** the ***key*** stored at the location with the value to be searched
  - **same as for storing a value in a hash table**
  - if match?
    - search time = O(1)
  - if not?
    - begin a *sequential* search
- three possible searching results
  - found the value
  - encounter a vacant location ⇨ indicating the value is not present
  - reach the end of the table ⇨ indicating the value is not present