Hashing

Unit-5 Lecture-20

Dr. Dillip Rout, Assistant Professor, Dept. of Computer Science and Engineering

Outline

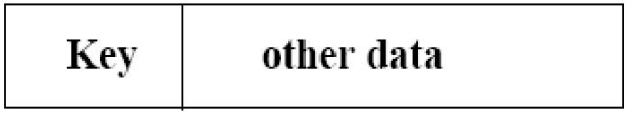
- Fundamentals of Hashing
- Hash Table/Map
- Hash Functions
- Collision Resolution
- Homework
- Conclusions

Fundamentals of Hashing

The Search Problem

- Find items with keys matching a given search key
  - Given an array A, containing n keys, and a search key x, find the index i such as x=A[i]
  - As in the case of sorting, a key could be part of a large record

example of a record



Applications

- Keeping track of customer account information at a bank
  - Search through records to check balances and perform transactions
- Keep track of reservations on flights
  - Search to find empty seats, cancel/modify reservations
- Search engine
  - Looks for all documents containing a given word

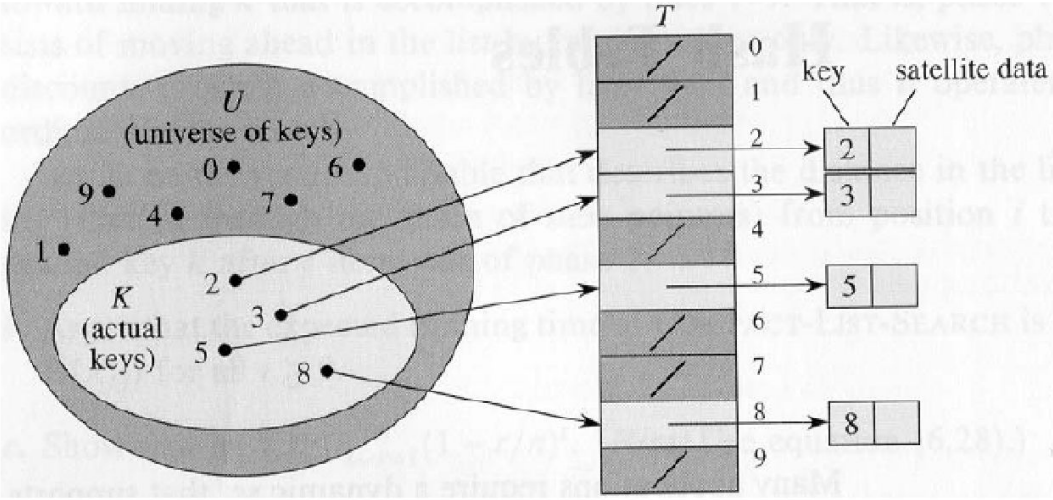
Special Case: Dictionaries

- **Dictionary** = data structure that supports mainly two basic operations: **insert** a new item and **return an item with a given key**
- Queries: return information about the set S:
  - Search (S, k)
  - Minimum (S), Maximum (S)
  - Successor (S, x), Predecessor (S, x)
- Modifying operations: change the set
  - Insert (S, k)

Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe $U = \{0, 1, \ldots, m - 1\}$
- Idea:
  - Store the items in an array, indexed by keys
- **Direct-address table** representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
  - For an element x with key k, a pointer to x (or x itself) 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

Direct Addressing (cont'd)



(insert/delete in O(1) time)

Operations

Alg.: DIRECT-ADDRESS-SEARCH(T, k)

    return T[k]

Alg.: DIRECT-ADDRESS-INSERT(T, x)

    T[key[x]] ← x

Alg.: DIRECT-ADDRESS-DELETE(T, x)

    T[key[x]] ← NIL

Comparing Different Implementations

- Implementing dictionaries using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered/unordered linked lists

	Insert	Search
direct addressing	O(1)	O(1)
ordered array	O(N)	O(lgN)
ordered list	O(N)	O(N)
unordered array	O(1)	O(N)
unordered list	O(1)	O(N)

Examples Using Direct Addressing

Example 1:

(i) Suppose that the keys are integers from 1 to 100 and that there are about 100 records

(ii) Create an array A of 100 items and store the record whose key is equal to i in $A[i]$

Example 2:

(i) Suppose that the keys are nine-digit social security numbers

(ii) We can use the same strategy as before but it very inefficient now: an array of 1 billion items is needed to store 100 records !!

- $|U|$ can be very large
- $|K|$ can be much smaller than $|U|$

Hash Tables

Hash Tables

- When K is much smaller than U, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to $|K|$
  - Can still get $O(1)$ search time, but on the average case, not the worst case

Hash Tables

Idea:

- Use a function h to compute the slot for each key
- Store the element in slot $h(k)$

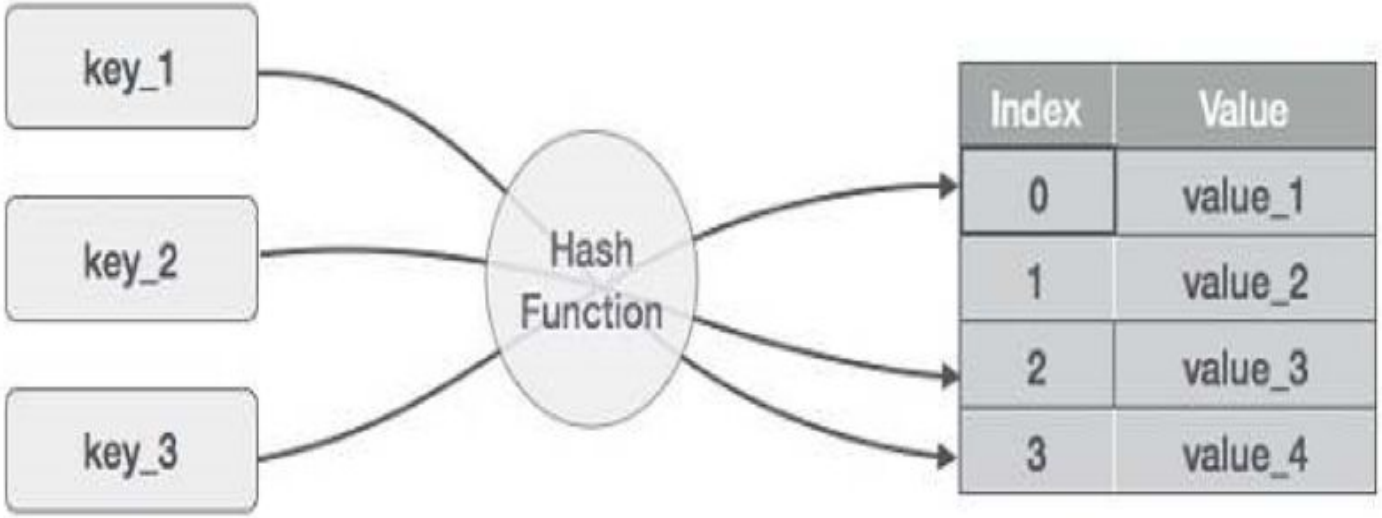
- A **hash function** h transforms a key into an index in a hash table $T[0...m-1]$:

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

- We say that k **hashes** to slot $h(k)$
- Advantages:
  - Reduce the range of array indices handled: m instead of $|U|$

Example: HASH TABLE

Continued…



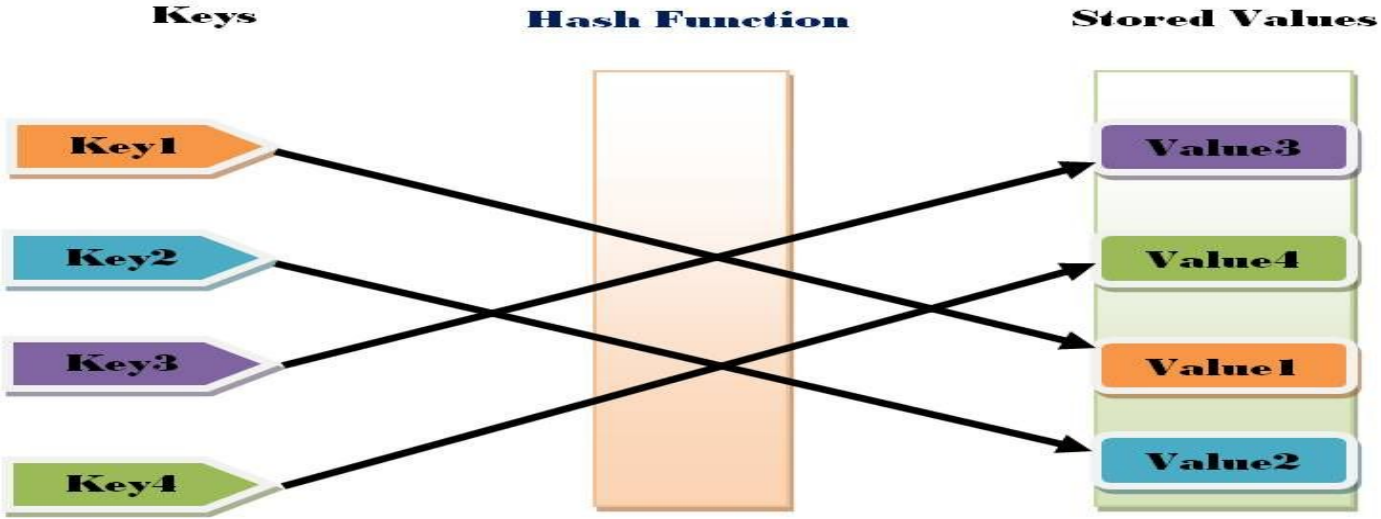
Example: HASH TABLES

Hash Functions

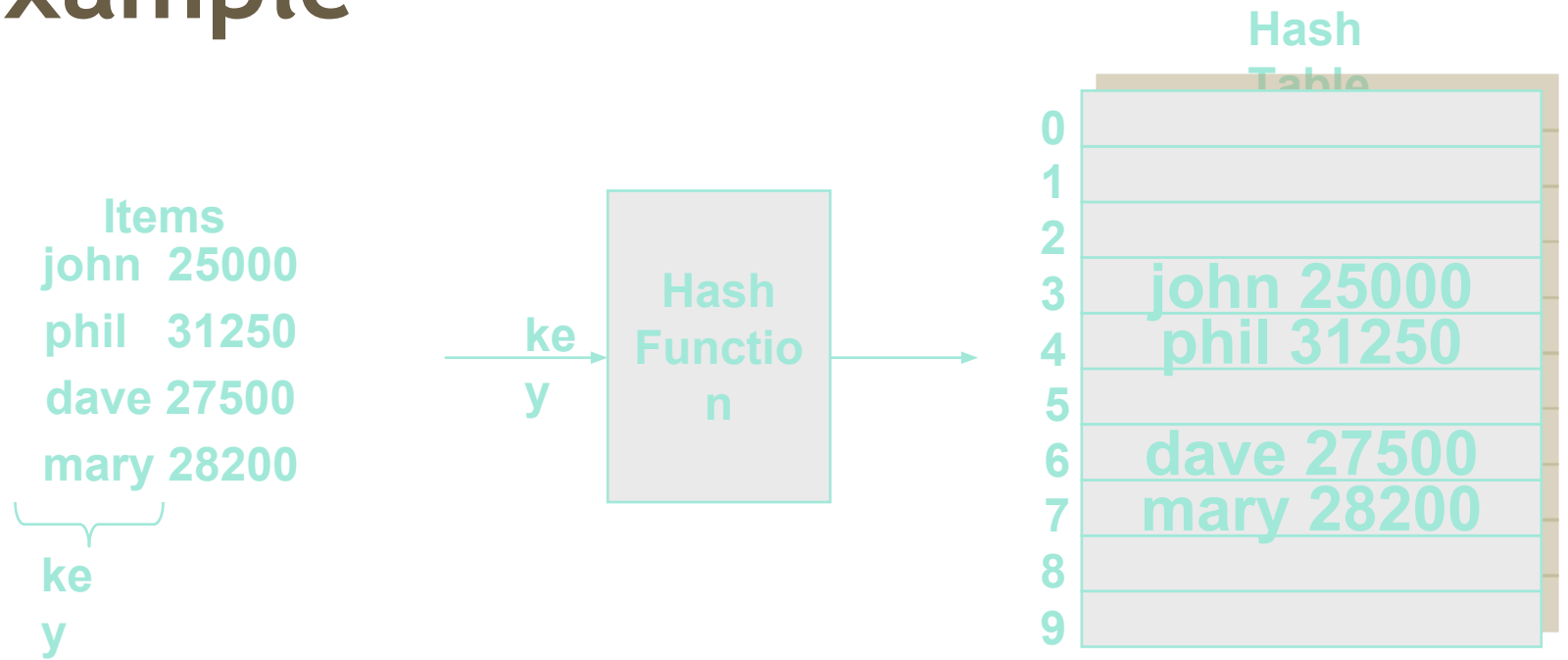
Choosing a Hash Function

- A hash function maps keys to small integers (buckets). An ideal hash function maps the keys to the integers in a random-like manner, so that bucket values are evenly distributed even if there are regularities in the input data.

- This process can be divided into two steps:



Example


Items
john 25000
phil 31250
dave 27500
mary 28200
key
key
Hash Function
Hash Table
0
1
2
3
4
5
6
7
8
9
john 25000
phil 31250
dave 27500
mary 28200

- The hash function:
  - ✔ must be simple to compute.
  - ✔ must distribute the keys evenly among the cells.

Different Approaches to generate hash function

1. Truncation Method
2. Mid Square Method
3. Folding Method
4. Modular Method

1. Truncation Method

Idea - Ignore a part of the key and use the remaining part directly as the index

Ex:1 If a hash table contains 999 entries at the most or 999 different key indexes may be kept, then a hash function may be defined such that from an eight digit integer 12345678, first, second and fifth digits from the right may be used to define a key index i.e. 478, which is the key position in the hash lookup table where this element will be inserted. Any other key combination may be used.

Ex:2 If students have an 9-digit identification number, take the last 3 digits as the table position

e.g. **925371622 becomes 622**

Registration number: last 3 digits will work???

2. Mid-Square Method

- In Mid-Square method, the key element is multiplied by itself to yield the square of the given key.
- If the hash table contains maximum 999 entries, then three digits from the result of square may be chosen as a hash key for mapping the key element in the lookup table.
- It generates random sequences of hash keys, which are generally key dependent.
- Mid Square method is a flexible method of selecting the hash key value.

2. Mid-Square Method - Example

If the input is the number 4567, squaring yields an 8-digit number, 20857489.

The middle two digits of this result are 57.

All digits of the original key value (equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value.

Thus, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

3. Folding Method

Idea- Partition the key into several parts and combine the parts in a convenient way (often addition or multiplication) to obtain the index.

- **Ex-1: An eight digit integer can be divided into groups of three, three and two digits (or any other combination) the groups added together and truncated if necessary to be in the proper range of indices.**
  Hence 12345678 maps to 123+456+78 = 657, since any digit can affect the index, it offers a wide distribution of key values.
- **Ex-2: Split a 9-digit number into three 3-digit numbers, and add them**
  e.g. 925371622 becomes 925 + 376 + 622 = 1923

4. Modular Method

- For mapping a given key element in the hash table, mod operation of individual key is calculated. The remainder denotes particular address position of each element.
- The result so obtained is divided by an integer, usually taken to be the size of the hash table to obtain the remainder as the hash key to place that element in the lookup table.
  - 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, e.g.,
      - power of 2
      - non-prime numbers

Example - The Modular Method

- 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

k	k mod 97	k mod 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

Collision Resolution

Collision

- Two or more keys hash to the same slot!!
- 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 is hard, even with a good hash function

Revisit Example 2

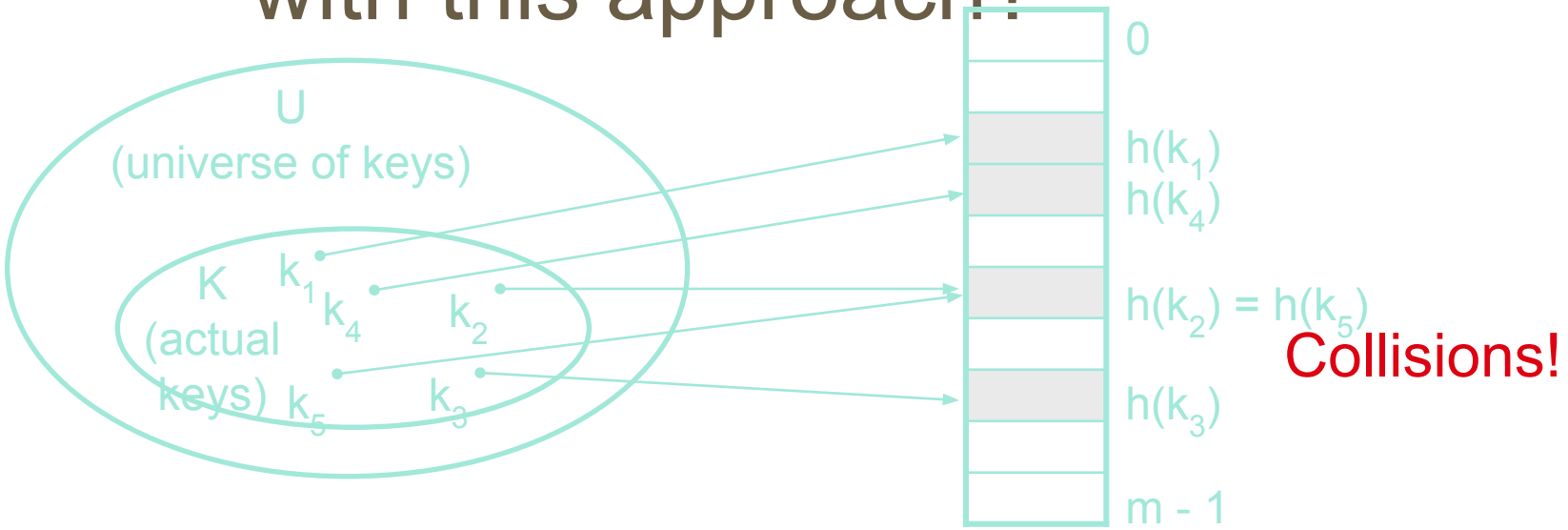
Suppose that the keys are nine-digit social security numbers

Possible hash function

$h(ssn) = sss \bmod 100$ (last 2 digits of ssn)

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

Do you see any problems with this approach?



Collisions

- Two or more keys hash to the same slot!!
- 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 is hard, even with a

Collision Resolution(Open Hashing)

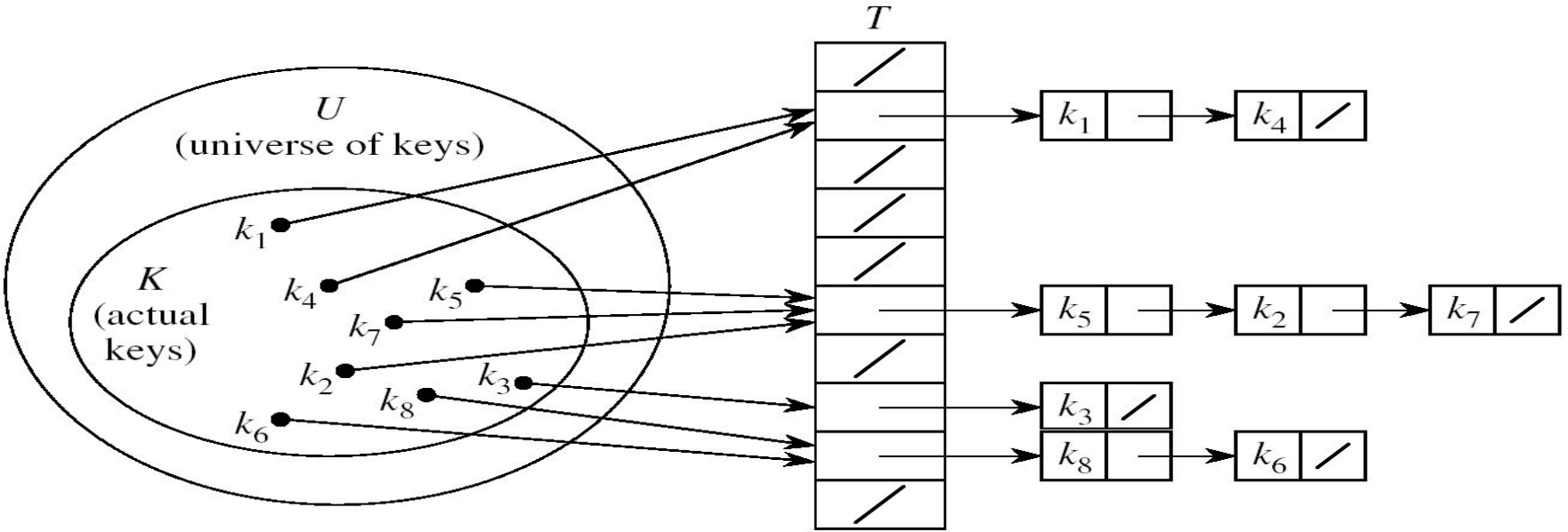
- If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it.
- There are several methods for dealing with this:
  - Separate chaining
  - Open addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

Handling Collisions

- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
- We will discuss chaining first, and ways to build "good" functions

Handling Collisions Using Chaining

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



Collision with Chaining - Discussion

- Choosing the size of the table
  - Small enough not to waste space
  - Large enough such that lists remain short
  - Typically 1/5 or 1/10 of the total number of elements
- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements

Insertion in Hash Tables

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

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

- Worst-case running time is $O(1)$
- Assumes that the element being inserted isn't already in the list
- It would take an additional search to check if it was

Deletion in Hash Tables

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

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

- Need to find the element to be deleted.
- Worst-case running time:
  - Deletion depends on searching the corresponding list

Searching in Hash Tables

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

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

- Running time is proportional to the length of the list of elements in slot h(k)

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
  - Worst-case time to search is $\Theta(n)$, plus time to compute the hash function

Analysis of Hashing with Chaining: Average Case

- Average case
  - depends on how well the hash function distributes the n keys among the m slots
- **Simple uniform hashing** assumption:
  - Any given element is equally likely to hash into any of the m slots (i.e., probability of collision Pr(h(x)=h(y)), is 1/m)
- Length of a list:

$$T[j] = n_j, \qquad j = 0, 1, \ldots, m - 1$$

- Number of keys in the table:

$$n = n_0 + n_1 + \cdots + n_{m-1}$$

Load Factor of a Hash Table

- Load factor of a hash table T:

  $\alpha = n/m$

  - n = # of elements stored in the table
  - m = # of slots in the table = # of linked lists
- α encodes the average number of elements stored in a chain
- α can be <, =, > 1

Case 1: Unsuccessful Search (i.e., item not stored in the table)

Theorem

An unsuccessful search in a hash table takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing

(i.e., probability of collision Pr(h(x)=h(y)), is 1/m)

Proof

- Searching unsuccessfully for any key k
  - need to search to the end of the list T[h(k)]
- Expected length of the list:
  - $E[n_{h(k)}] = \alpha = n/m$
- Expected number of elements examined in an unsuccessful search is α
- Total time required is: $\Theta(1+\alpha)$

Case 2: Successful Search

Successful search: $\Theta(1 + \frac{a}{2}) = \Theta(1 + a)$ time on the average

(search half of a list of length a plus $O(1)$ time to compute $h(k)$)

Analysis of Search in Hash Tables

- If m (# of slots) is proportional to n (# of elements in the table):
- n = O(m)
- α = n/m = O(m)/m = O(1)

⇒ Searching takes constant time on average

Hash Functions

- A hash function transforms a key into a table address
- **What makes a good hash function?**

  (1) Easy to compute

  (2) Approximates a random function: for every input, every output is equally likely (simple uniform hashing)

- In practice, it is very hard to satisfy the simple uniform hashing property
  - i.e., we don't know in advance the probability distribution that keys are drawn from

Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as `pt` and `pts` should hash to different slots
- **Derive a hash value that is independent from any patterns that may exist in the distribution of the keys**

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, e.g.,
    - power of 2
    - non-prime numbers

Example - The Division Method

- If $m = 2^p$, then $h(k)$ is just the least significant p bits of k
  - $p = 1 \Rightarrow m = 2$ $\{0, 1\}$
    $\Rightarrow h(k) =$, least significant 1 bit of k
  - $p = 2 \Rightarrow m = 4$ $\{0, 1, 2, 3\}$
    $\Rightarrow h(k) =$, least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2
  - Column 2:

$k \bmod 97$

$k \bmod 100$

	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

8

The Multiplication Method

Idea:

- Multiply key k by a constant A, where $0 < A < 1$
- Extract the fractional part of kA
- Multiply the fractional part by m
- Take the floor of the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

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

fractional part of $kA = kA - \lfloor kA \rfloor$

- **Disadvantage:** Slower than division method

Example – Multiplication Method

- The value of m is not critical now (e.g., $m = 2^p$)

assume $m = 2^3$

```
     .101101 (A)
      110101 (k)
--------------
1001010.0110011 (kA)
```

discard: 1001010

shift .0110011 by 3 bits to the left

011.0011

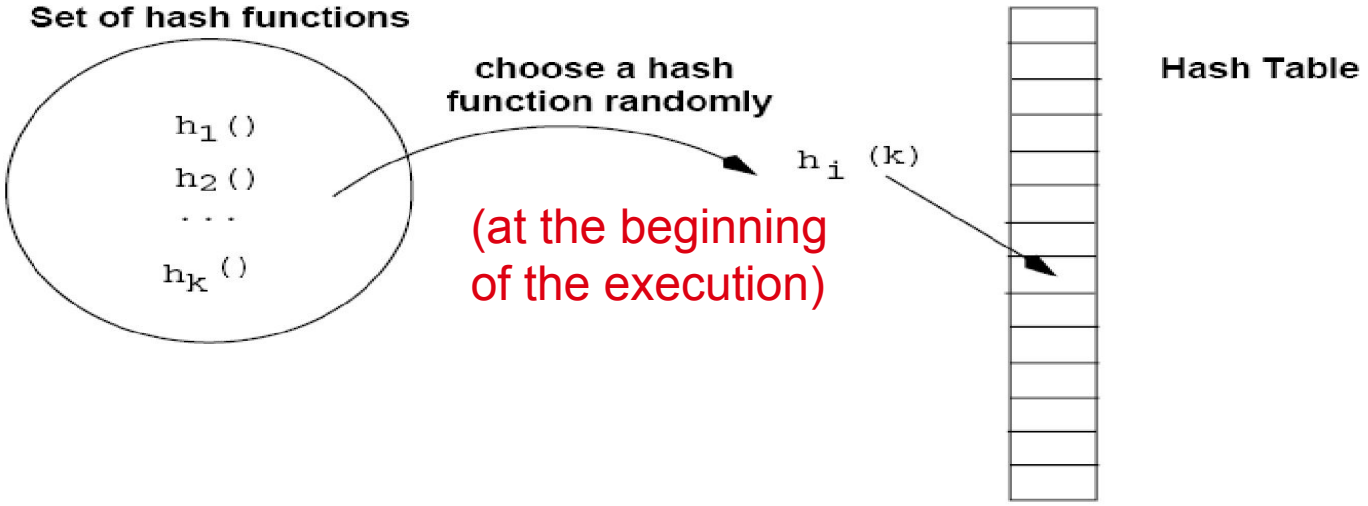
take integer part: 011

thus, h(110101)=011

Universal Hashing

- In practice, keys are not randomly distributed
- Any fixed hash function might yield Θ(n) time
- Goal: hash functions that produce random table indices irrespective of the keys
- Idea:
  - Select a hash function at random, from a designed class of functions at the beginning of the execution

Universal Hashing



Definition of Universal Hash Functions

H={h(k): U□(0,1,..,m-1)}

H is said to be universal if

$$\text{for } x \neq y,\ |(\mathbf{h() \in H: h(x)=h(y)}|=|\mathbf{H}|/\mathbf{m}$$

(notation: |H|: number of elements in H - cardinality of H)

How is this property useful?

- What is the probability of collision in this case ?

  It is equal to the probability of choosing a function $h \in U$ such that $x \neq y$ --> $h(x) = h(y)$ which is

Pr(h(x)=h(y))= $\frac{|H|/m}{|H|} = \frac{1}{m}$

Universal Hashing – Main Result

With universal hashing the chance of collision between distinct keys k and l is no more than the $1/m$ chance of collision if locations $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m - 1\}$

Designing a Universal Class of Hash Functions

- Choose a prime number p large enough so that every possible key k is in the range $[0 \ldots p - 1]$

  $Z_p = \{0, 1, \ldots, p - 1\}$ and $Z_p^* = \{1, \ldots, p - 1\}$

- Define the following hash function

  $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m,$

  $\forall\ a \in Z_p^*$ and $b \in Z_p$

- The family of all such hash functions is

  $\mathcal{H}_{p,m} = \{h_{a,b} : a \in Z_p^* \text{ and } b \in Z_p\}$

The class $\mathcal{H}_{p,m}$ of hash functions is universal

Example: Universal Hash Functions

E.g.: $p = 17, m = 6$

$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$

$h_{3,4}(8) = ((3 \cdot 8 + 4) \bmod 17) \bmod 6$

$= (28 \bmod 17) \bmod 6$

$= 11 \bmod 6$

$= 5$

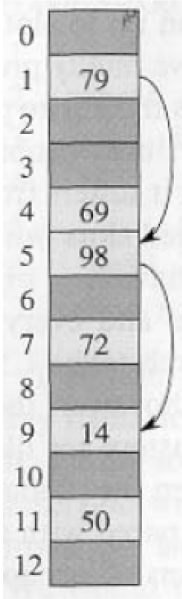
Advantages of Universal Hashing

- Universal hashing provides good results on average, independently of the keys to be stored
- Guarantees that no input will always elicit the worst-case behavior
- Poor performance occurs only when the random choice returns an inefficient hash function – this has small probability

Open Addressing

- If we have enough contiguous memory to store all the keys (m > N) ⇒ store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - Insertion: if a slot is full, try another one, until you find an empty one
  - Search: follow the same sequence of probes
  - Deletion: more difficult ... (we'll see why)
- Search time depends on the length of the

e.g., insert 14



Generalize hash function notation:

- A hash function contains two arguments now:

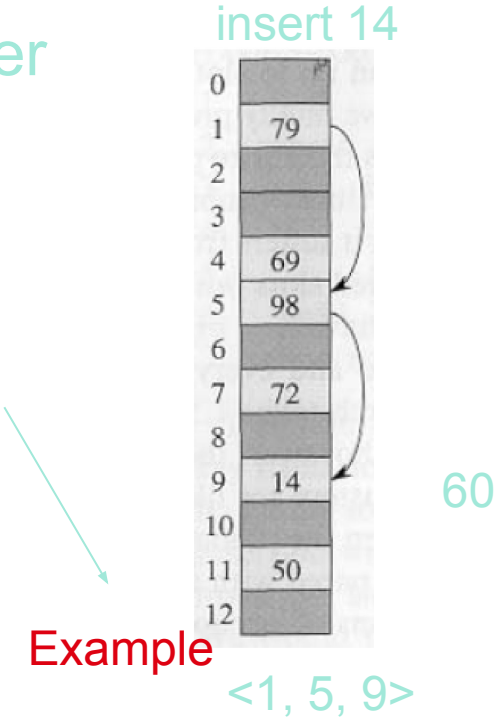
  (i) Key value, and (ii) Probe number

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

- Probe sequences

  $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$

  - Must be a permutation of $\langle 0,1,\ldots,m-1 \rangle$
  - There are $m!$ possible permutations



Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

- Note: None of these methods can generate more than m^2 different probing sequences!

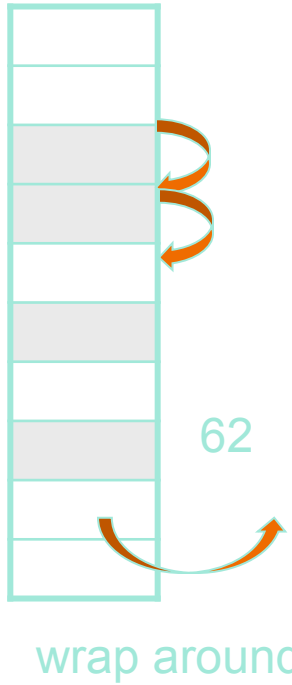
Linear probing: Inserting a key

- Idea: when there is a collision, check the next available position in the table (i.e., probing)

$$h(k,i) = (h_1(k) + i) \bmod m$$
$$i=0,1,2,...$$

- First slot probed: $h_1(k)$
- Second slot probed: $h_1(k) + 1$
- Third slot probed: $h_1(k)+2$, and so on

probe sequence: < h1(k), h1(k)+1 , h1(k)+2 , ...>



Linear probing: Searching for a key

- Three cases:
  - (1) Position in table is occupied with an element of equal key
  - (2) Position in table is empty
  - (3) Position in table occupied with a different element
- Case 2: probe the next higher index until the element is found or an empty position is found
- The process wraps around to the

Linear probing: Deleting a key

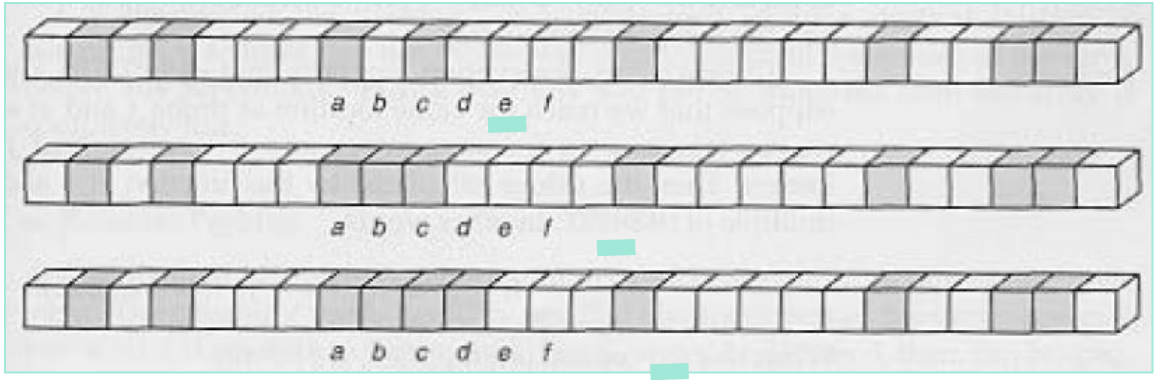
- Problems
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion
- Searching will be able to find all the

Primary Clustering Problem

- Some slots become more likely than others
- Long chunks of occupied slots are created

⇒ search time increases!!

initially, all slots have probability 1/m



Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Quadratic probing

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

i=0,1,2,...

- Clustering problem is less serious but still an issue (*secondary clustering*)

- How many probe sequences quadratic probing generate ? m

(the initial probe position determines the 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

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

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on ...
- Advantage: avoids clustering
- Disadvantage: harder to delete an element
- Can generate m^2 probe sequences maximum

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:

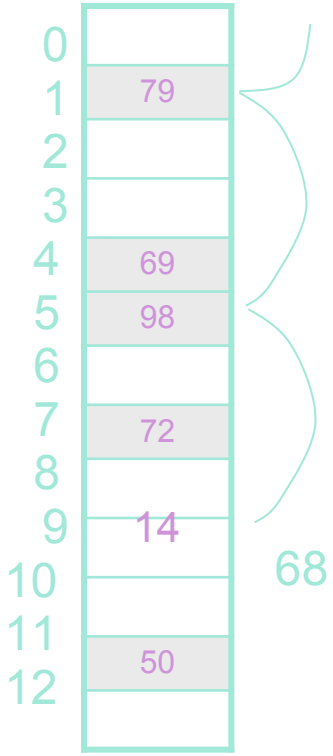
$h_1(14,0) = 14 \bmod 13 = 1$

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

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

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

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



Analysis of Open Addressing

- Ignore the problem of clustering and assume that all probe sequences are equally likely

Unsuccessful retrieval:

Prob(*probe hits an occupied cell*)=a (load factor)

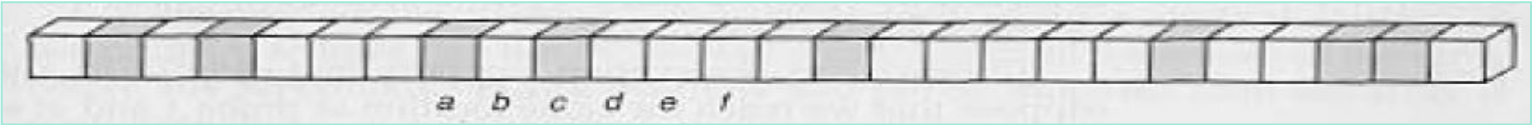
Prob(*probe hits an empty cell*)=$1-a$

probability that a probe terminates in 2 steps: $a(1-a)$

probability that a probe terminates in k steps: $a^{k-1}(1-a)$

What is the average number of steps in a probe ?

$$E(\#steps) = \sum_{k=1}^{m} ka^{k-1}(1-a) \le \sum_{k=0}^{\infty} ka^{k-1}(1-a) = (1-a)\frac{1}{(1-a)^2} = \frac{1}{1-a}$$



Analysis of Open Addressing (cont'd)

Successful retrieval:

$$E(\#steps) = \frac{1}{a} ln(\frac{1}{1-a})$$

Example (similar to **Exercise 11.4-4, page 244**)

Unsuccessful retrieval:

a=0.5 E(#steps) = 2

a=0.9 E(#steps) = 10

Successful retrieval:

a=0.5 E(#steps) =

Homework

Conclusions