Hashing

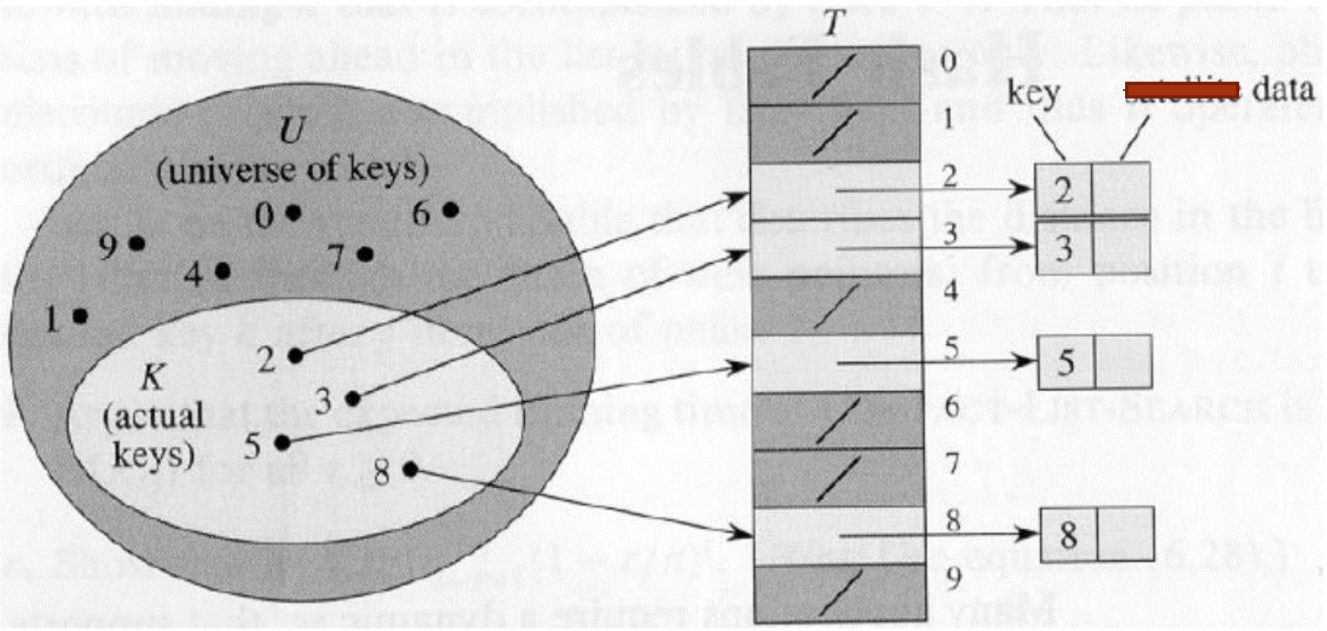
- The sequential search algorithm takes time proportional to the data size, i.e, `O(n)`.
- Binary search improves on liner search reducing the search time to `O(log n)`.
- With a BST, an `O(log n)` search efficiency can be obtained; but the worst-case complexity is `O(n)`.
  - To guarantee the `O(log n)` search time, BST height balancing is required (i.e., AVL trees).

Hashing

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.
  - A linked list implementation would take `O(n)` time.
  - A height balanced tree would give `O(log n)` access time.
  - Using an array of size 100,000 would give `O(1)` access time but will lead to a lot of space wastage.

Direct Addressing

Here, 6 memory locations are wasted.



(insert/delete in O(1) time)

Hashing

Is there some way that we could get `O(1)` access without wasting a lot of space?

- The answer is hashing.
  - Constant time per operation (on the average)
  - Like an array, come up with a function to map the large range into one which we can manage.

Basic Idea

- Use *hash (or hashing) function* to map hash key into hash address (location) in a *hash table*

  Hash Function H: K -> L
- If Student A has ID(Key) *k* and *h* is hash function, then *A's Details* is stored in position *h(k)* of table
- To search for A, compute *h(k)* to locate position. If no element, hash table does not contain *A*.

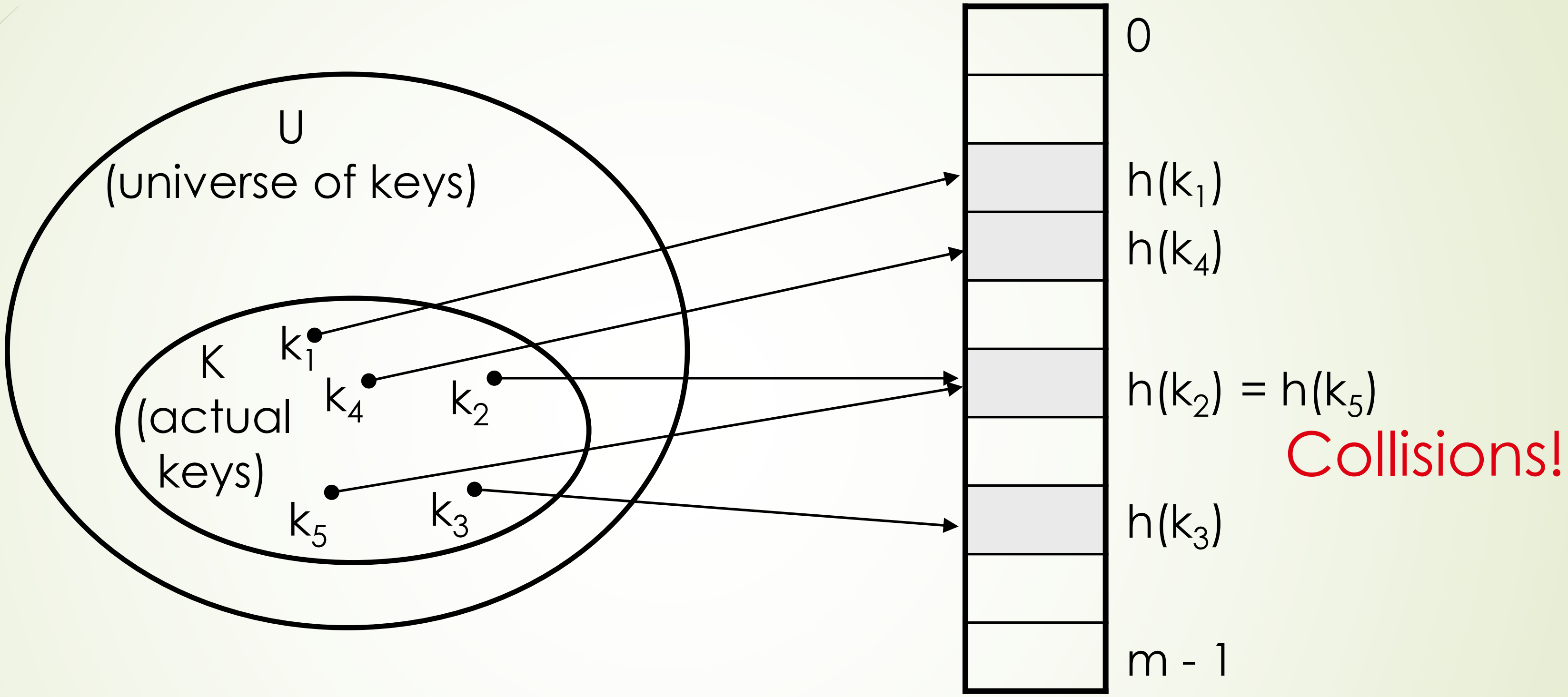
Example

Let keys be ID of 100 students And ID in form of like 345610.

Now, we decided to take A[100]
And, Hash function is , say , LAST TWO DIGIT

So, 103062 will go to location 62 And same if some one have 113062 Then again goes to the location 62
THIS EVENT IS CALLED **COLLISION**

Collisions



U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_2
k_5
k_3
0
$h(k_1)$
$h(k_4)$
$h(k_2) = h(k_5)$
Collisions!
$h(k_3)$
m - 1

Collisions

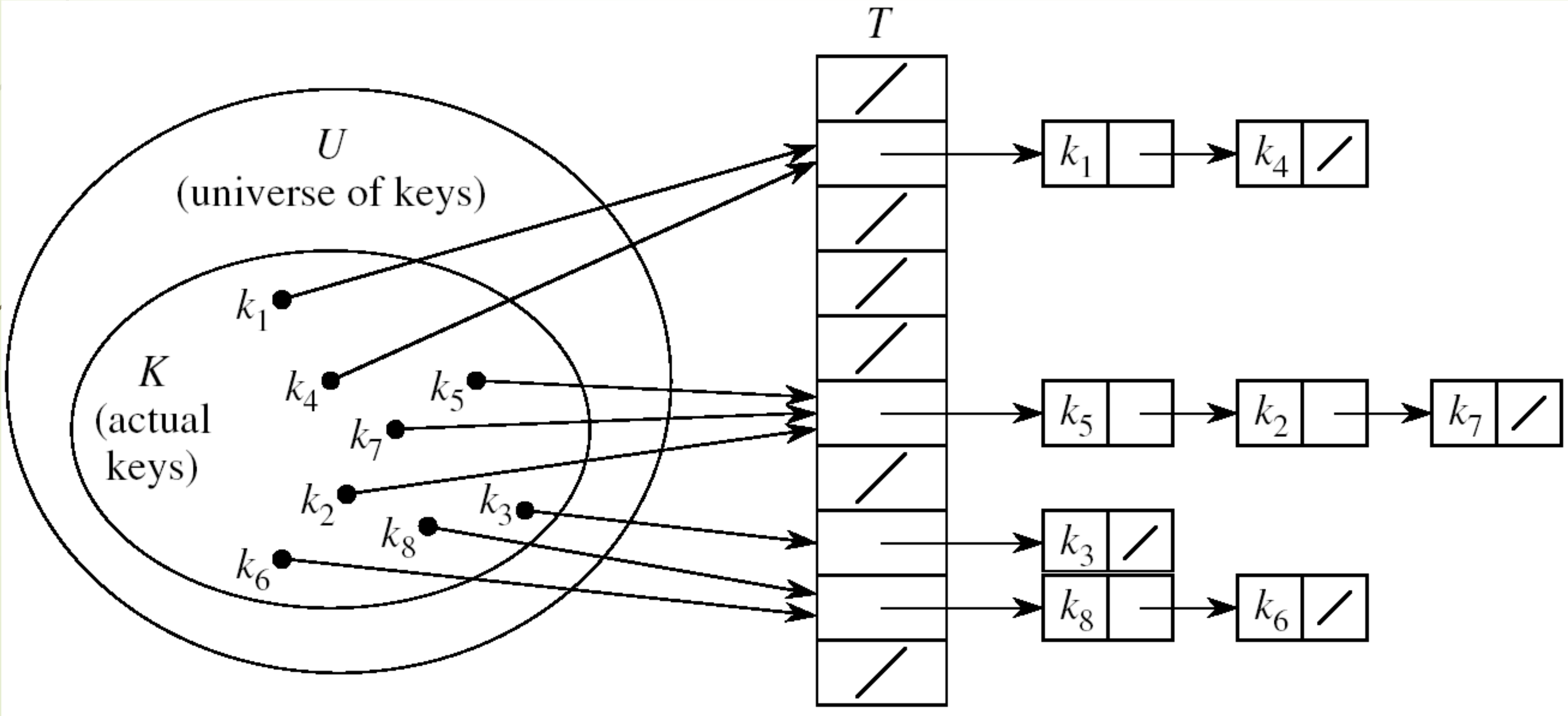
- Two or more keys hash to the same location
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function

Collision Resolution

- Methods:
  - Separate Chaining (open hashing)
  - Open addressing (closed hashing)
    - 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



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

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

- 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 already inserted

Deletion in Hash Tables

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

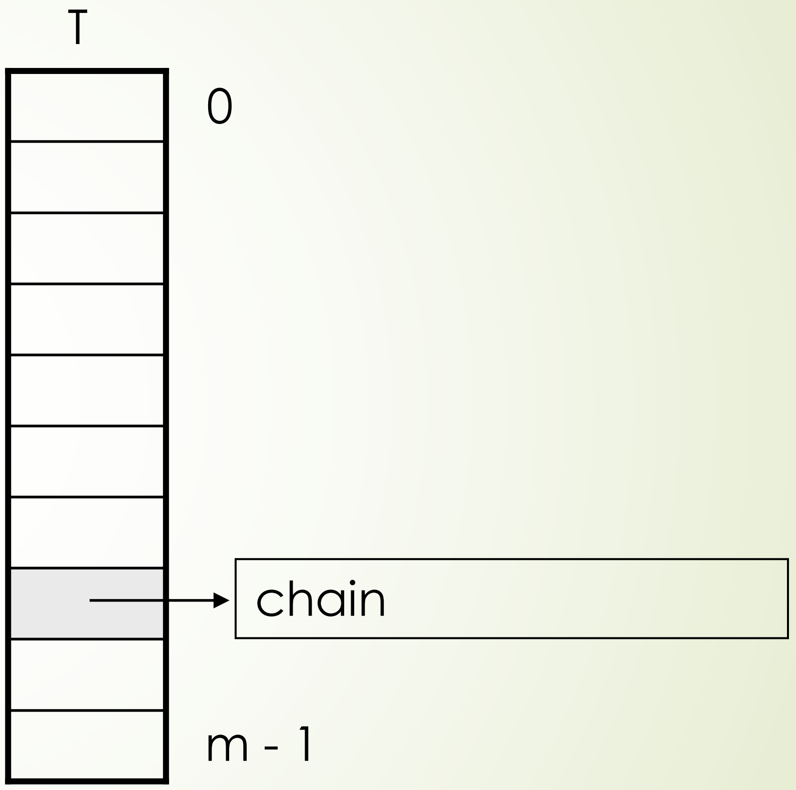
Searching in Hash Tables

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

- Running time is proportional to the length of the list of elements at location **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:

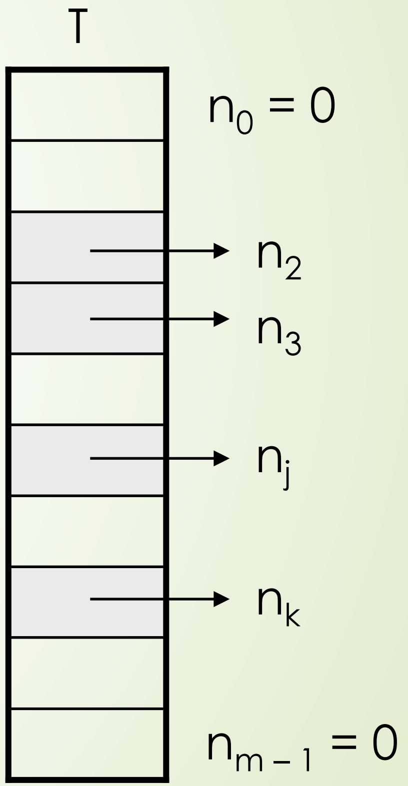
$$T[j] = n_j, \qquad j = 0, 1, \ldots, m - 1$$

- Number of keys in the table:

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

- Average value of n_j:

$$E[n_j] = \alpha = n/m$$

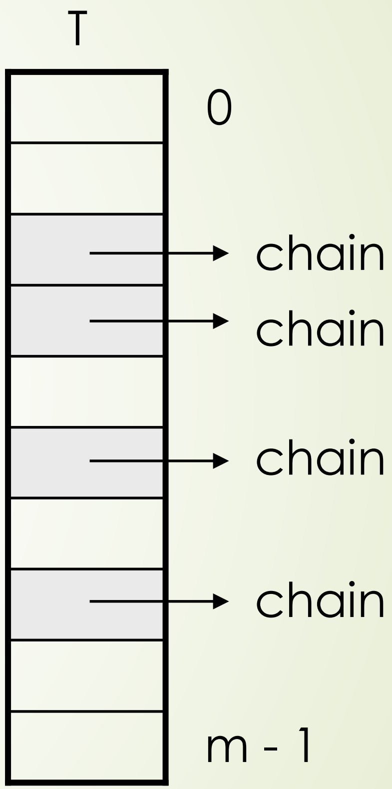


Load Factor of a Hash Table

- Load factor of a hash table T:

  $\alpha = n/m$

  - n = # of elements stored in the table
  - m = # of locations 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: $\Theta(1+\alpha)$
  - $\Theta(1)$ (for computing the hash function) + α →

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):
- $n = O(m)$
- $a = n/m = O(m)/m = O(1)$

$\Rightarrow$ Searching takes constant time on average

Hash Functions

- A hash function transforms a hash key into a hash 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
  - Hash keys such as **199** and **499** should hash to different slots

The Division Method

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

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

- **m is usually chosen to be a prime number or a number without small divisors to minimize the number of collisions**
- **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
  - $p = 1 \Rightarrow m = 2$

    $\Rightarrow h(k) = \{0, 1\}$, 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:

- 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

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

- **Disadvantage:** Slower than division method
- **Advantage:** Value of **m** is not critical, e.g., typically 2^p

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

The Midsquare Method

- **Idea:**

Key k is squared.

$h(k) = l$

l is obtained by deleting digits from both ends of k^2

Same positions of k^2 are used for all the keys

K	3205	7148	2345
K2	10272025	51093904	5499025
H(k)	72	93	99

The Folding Method

- **Idea:**

Key k is partitioned into a number of parts k_1, $k_2,\ldots,k_r$ Where each part except possibly the last has same number of digits as the required address

$$h(k) = k_1 + k_2 + \ldots + k_r$$

Where leading-digit carries are ignored, if any

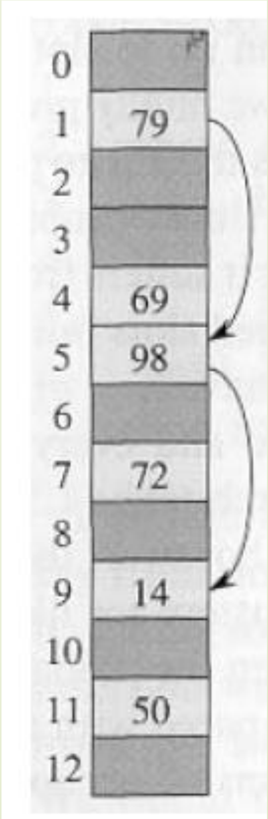
Sometimes even numbered parts are reversed

K	3205	7148	2345
	32+05	71+48	23+45
H(k)	37	19	68
H(k)	82	55	77 second part is reversed

Open Addressing

- If we have enough contiguous memory to store all the keys (m > N)
  $\Rightarrow$ store the keys in the table itself
  - It is called "open" because the address where key k is stored also depends on keys already stored in hash table along with h(k)
  - It is also called closed hashing
- No need to use linked lists anymore
  - Hashing with chaining is called open hashing
- 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
- Search time depends on the length of the probe sequence!

e.g., insert 14



Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

Linear probing: Inserting a key

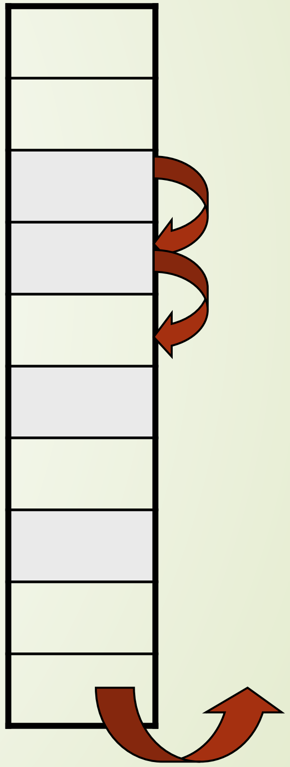
- Idea: when there is a collision, check the next available position in the table (i.e., probing)

(h(k) + i) **mod** m, i=0,1,2,...

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

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

- The process wraps around to the beginning of the table



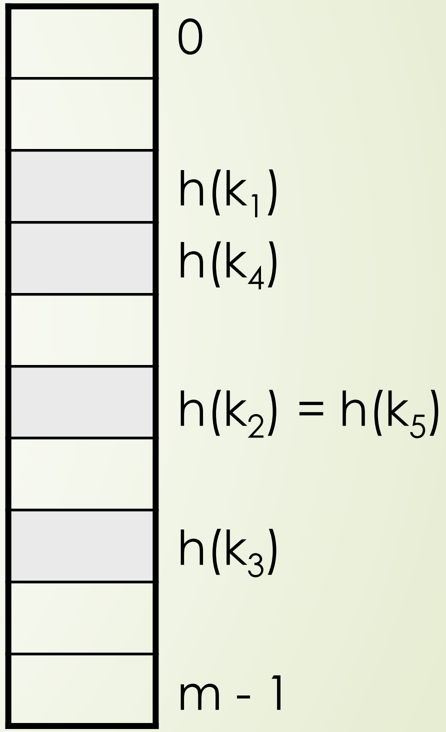
wrap around

Linear Probing Example

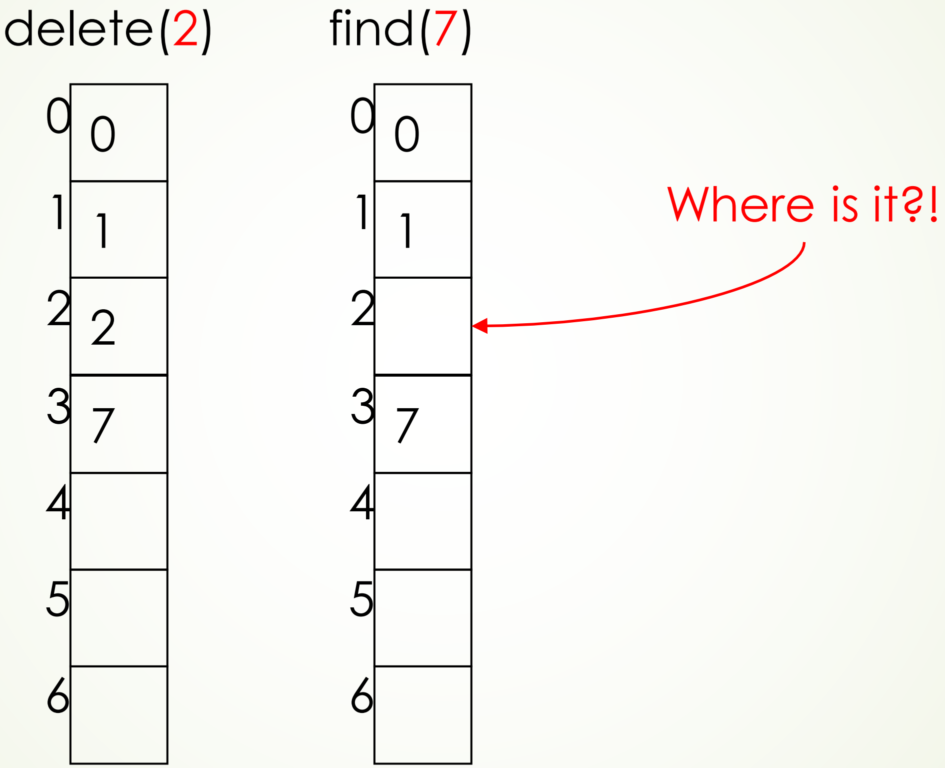
	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2
0	14	14	14	14
1		8	8	8
2			21	12
3				2
4				
5				
6				
probes:	1	1	3	2

Linear probing: Searching for a key

- probe the next higher index until the element is found (successful search) or an empty position is found (unsuccessful search)
- The process wraps around to the beginning of the table



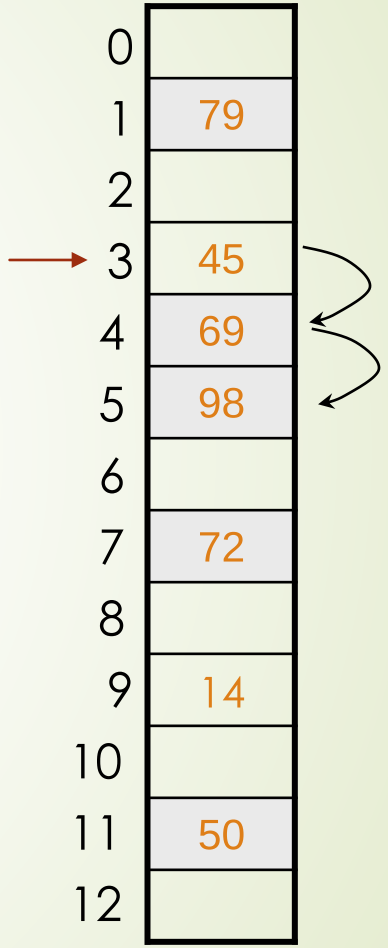
Deletion in Closed Hashing



What should we do instead?

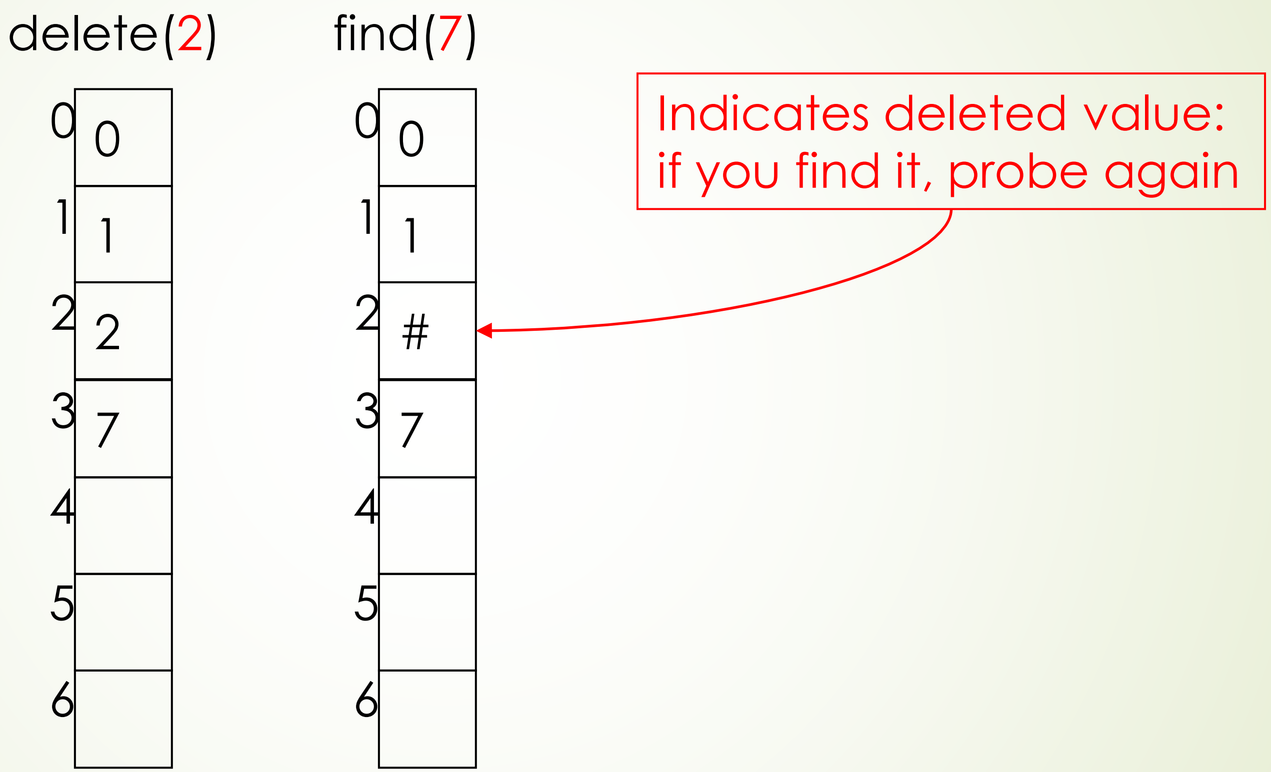
Deleting a key in Closed Hashing

- 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 keys



45, 69 and 98 are hashed to same hash address 3
Delete 69

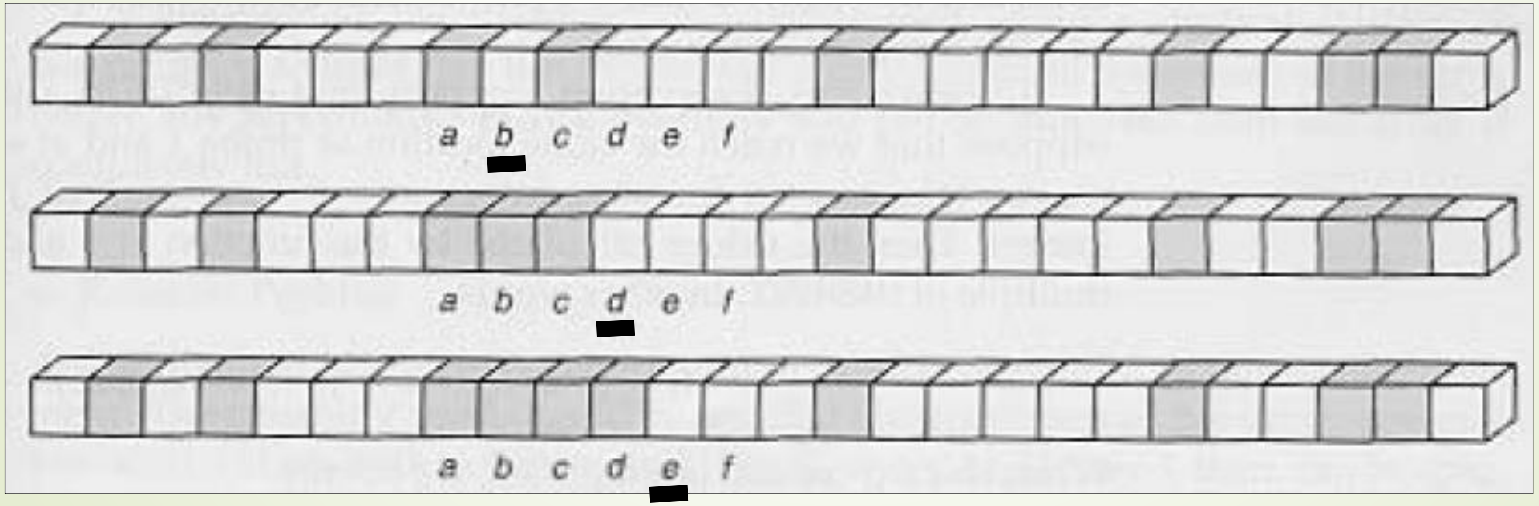
Lazy Deletion



Linear probing: Primary Clustering Problem

- Some slots become more likely than others
- clusters grow when keys hash to values close to each other
  - if a bunch of elements hash to the same area of the table, they mess each other up! (Even though the hash function isn't producing lots of collisions!)
- Long chunks of occupied slots are created

  $\Rightarrow$ search time increases!!



Slot b:
2/m

Slot d:
4/m

Slot e:
5/m

initially, all slots have probability 1/m

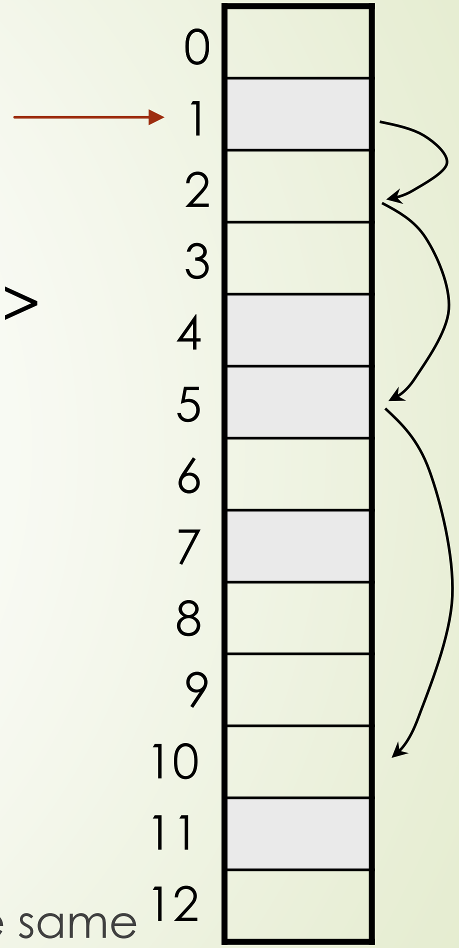
Quadratic probing: Inserting a key

- Idea: when there is a collision, check the next available position in the table

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

probe sequence: < h(k), h(k)+1 , h(k)+4 , ...>

- First slot probed: h(k)
- Second slot probed: h(k) + 1
- Third slot probed: h(k)+4, and so on
- The process wraps around to the beginning of the table
- Clustering problem is less serious
  - But it is still an issue (secondary clustering)
    - multiple keys hashed to the same spot all follow the same probe sequence.



Quadratic Probing Example

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2
0	14	14	14	14
1		8	8	8
2				2
3				
4			21	21
5				
6				
probes:	1	1	3	1

Problem With Quadratic Probing

	insert(14)	insert(8)	insert(21)	insert(2)	insert(7)
	14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2	7%7 = 0
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
probes:	1	1	3	1	??

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

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

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

Double Hashing: Example

$h_1(k) = k \bmod 13$

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

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

- Insert key 14:

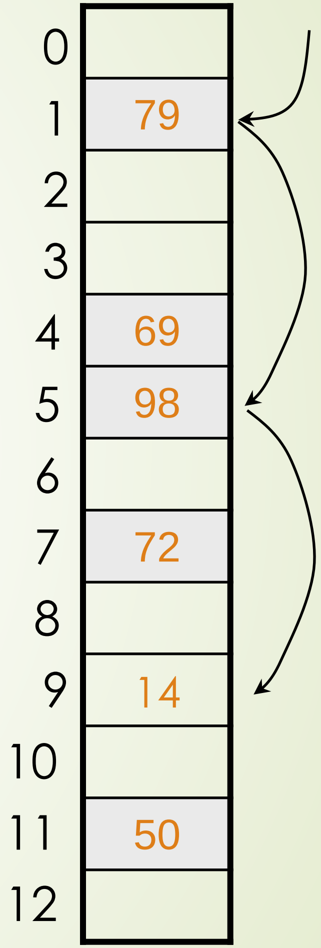
$h_1(14) = 14 \bmod 13 = 1$

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

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

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

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



Double Hashing Example

	insert(14) $14\%7 = 0$	insert(8) $8\%7 = 1$	insert(21) $21\%7 = 0$, $5-(21\%5)=4$	insert(2) $2\%7 = 2$	insert(7) $7\%7 = 0$, $5-(21\%5)=4$
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
probes:	1	1	2	1	??

Double Hashing Example

	insert(14)	insert(8)	insert(21)	insert(2)	insert(56)
	14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2	56%7 = 0
			5-(21%5)=4		5-(56%5)=4
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					56
6					
probes:	1	1	2	1	4

Theoretical Results

- Let α = n/m
the load factor: average number of keys per array index
- Analysis is probabilistic, rather than worst-case

Expected Number of Probes

	Not found	***found***
Chaining	$1+\alpha$	$1+\frac{\alpha}{2}$
Linear Probing	$\frac{1}{2}+\frac{1}{2(1-\alpha)^2}$	$\frac{1}{2}+\frac{1}{2(1-\alpha)}$
Double Hashing	$\frac{1}{(1-\alpha)}$	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

Analysis of Double Hashing

Successful retrieval:

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

Example

Unsuccessful retrieval:

α =0.5 E(#steps) = 2

α =0.9 E(#steps) = 10

Successful retrieval:

α =0.5 E(#steps) = 3.387

α =0.9 E(#steps) = 3.670

Rehashing

- An insert using Closed Hashing *cannot* work with a load factor of 1 or more.
  - Quadratic probing can *fail* if $\alpha > ½$
  - Linear probing and double hashing *slow* if $\alpha > ½$
  - Lazy deletion never frees space
- Separate chaining becomes slow once $\alpha > 1$
  - Eventually becomes a linear search of long chains
- Solution: REHASH!

Rehashing Example

Separate chaining

$h_1(x) = x \bmod 5$ rehashes to $h_2(x) = x \bmod 11$

$\alpha = 1$

0	1	2	3	4
25		37 52	83 98	

$\alpha = 5/11$

0	1	2	3	4	5	6	7	8	9	10
			25	37		83		52		98

Case Study

- Spelling dictionary
  - 50,000 words
  - static
  - arbitrary preprocessing time
- Goals
  - fast spell checking
  - minimal storage

- Practical notes
  - almost all searches are successful
  - words average about 8 characters in length
  - 50,000 words at 8 bytes/word is 400K
  - pointers are 4 bytes

Mostly correct input

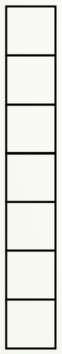
Solutions

- sorted array + binary search
- separate chaining
- open addressing + linear probing

Storage

- words are strings

Array +
binary search



n words

Separate chaining

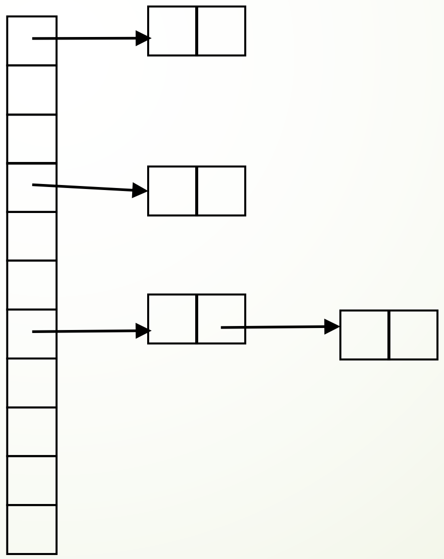
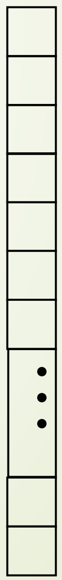


table size + n pointers, n words
= n/ α + n pointers, n words

Closed hashing



n/ α words

Analysis

50K words, 4 bytes @ pointer

- Binary search
  - storage: n words = 400K
  - time: $\log_2 n \leq 16$ probes per access, worst case
- Separate chaining - with $\alpha = 1$
  - storage: n/α + n pointers + n words = 200K+200K+400K = 800KB
  - Time (success): $1 + \alpha/2$ probes per access on average = 1.5
- Closed hashing - with $\alpha = 0.5$
  - storage: n/α words = 400K + 400K = 800K
  - Time (LP Success) : $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$ probes per access on average = 1.5