Chapter 11

Hashing

Credit: Dr. George Bebis

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 that $x=A[i]$
  - a key could be part of a large record.

example of a record

Key	other data

Applications

- **Banking**: Keeping track of customer account information
  - Search through records to check balances and perform transactions
- **Flight reservations**: Search to find empty seats, cancel/modify reservations
- **Search engine**: Looks for all documents containing a given word

Dictionaries

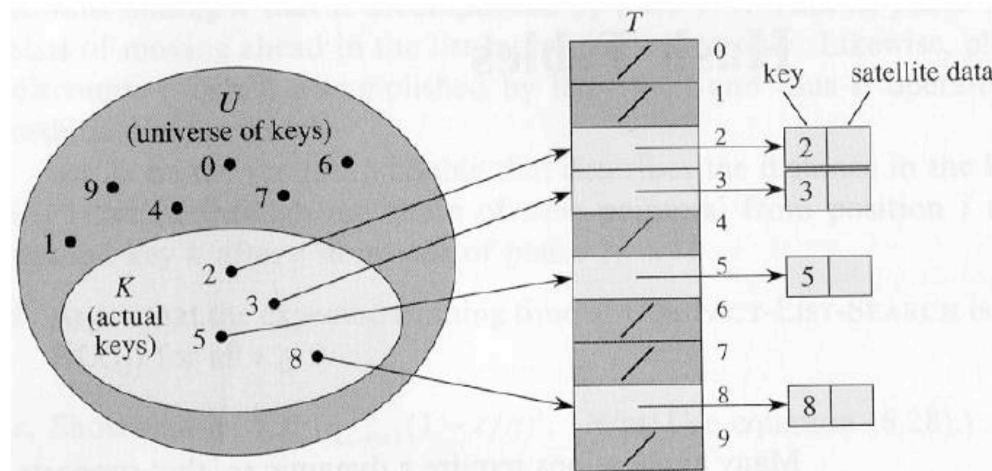
- **Dictionary** = data structure that supports the basic operations: insert, delete and search an item (based on a given key)
- **Queries**: return information about the set S:
  - Search (S, k)
  - Minimum (S), Maximum (S)
  - Successor (S, x), Predecessor (S, x)
- **Operations that modify** a given set
  - Insert (S, k)
  - Delete (S, k) – not very often

Direct Addressing

- If key values are **distinct** and drawn from a **universe** U = {0, 1, . . . , m - 1}

You may use **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] points to a NIL



(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

- **Running time for these operations: *O*(1)**

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:

- Suppose that the keys are 9-digit Social Security Numbers
- We can use the same strategy as before but it will be very inefficient. An array of 1 Billion items is needed to store 100 records !!

- $|U|$ (size of the entire key universe) can be very large
- $|K|$ (actual set of keys) can be much smaller than $|U|$

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|
- and still get $O(1)$ search time on average (not the worst case)

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]. $h : U \rightarrow \{0, 1, . . . , m - 1\}$
- We say that k **hashes** to slot h(k)
- Advantages:
  - Reduce the range of array indices handled: m instead of |U|

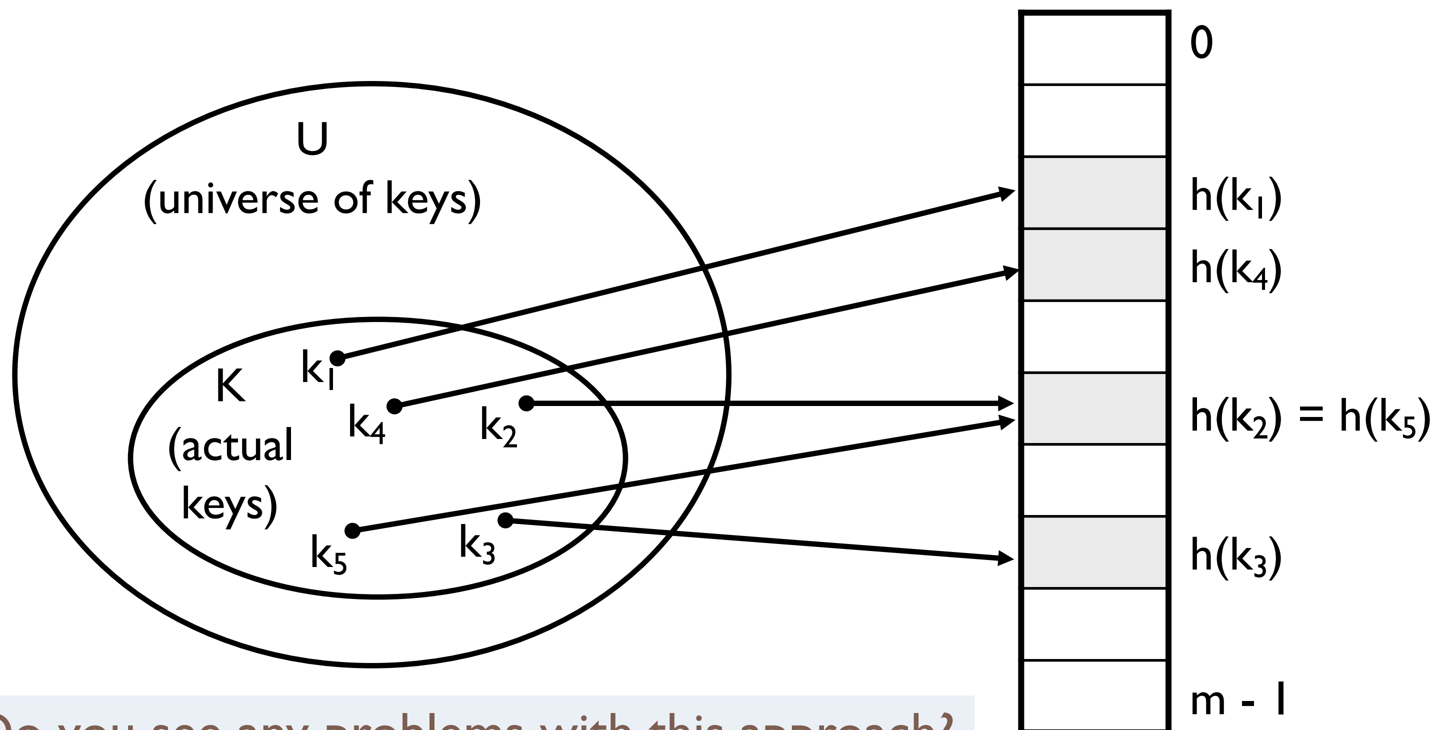
Revisit Example 2

Suppose that the keys are 9-digit Social Security Numbers

One possible hash function:

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

e.g. If ssn = 108 334 198 then $h(108334198) = 98$



Do you see any problems with this approach?

Collisions can occur!

Collisions

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

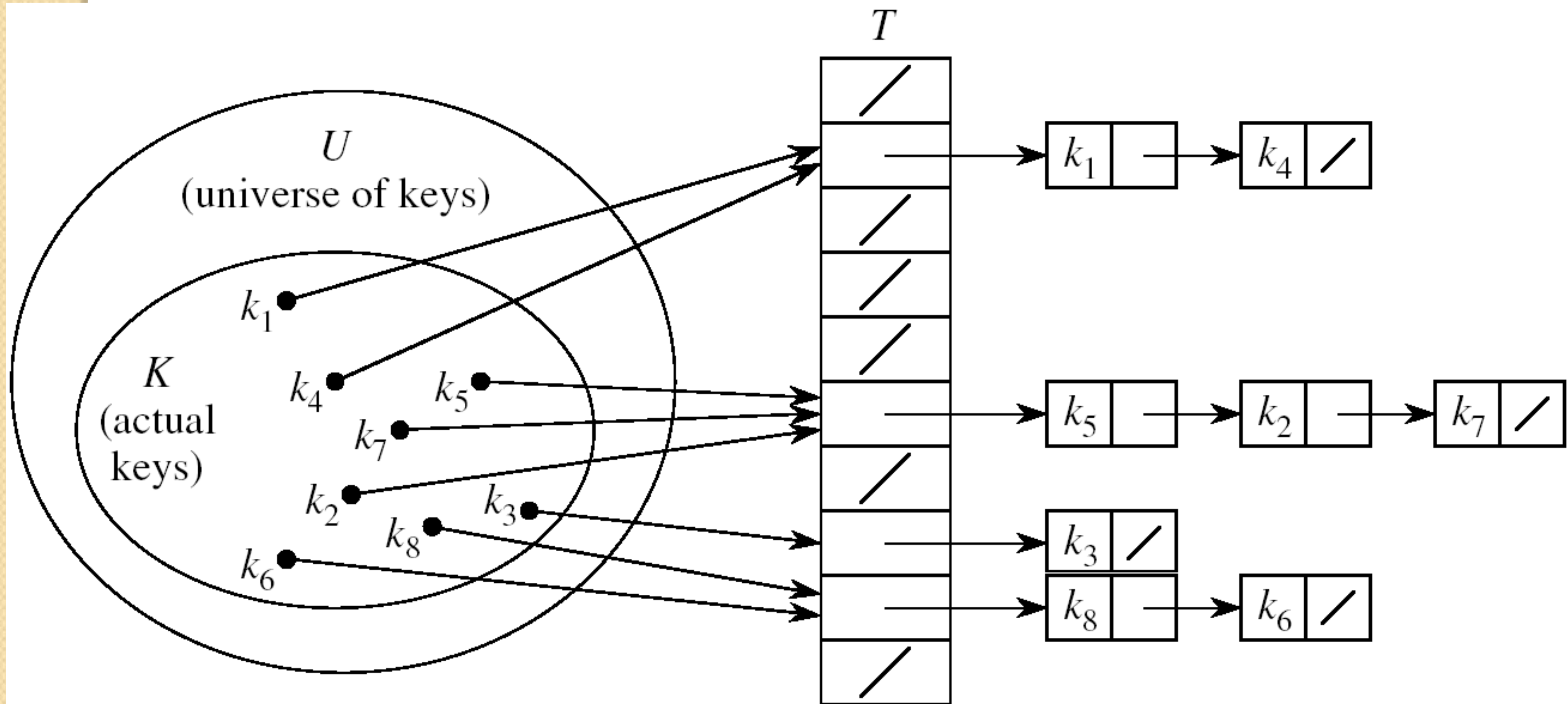
Handling Collisions:

- **Chaining**
- **Open addressing**
  - Linear probing
  - Quadratic probing
  - Double hashing

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



Operations in Chained Hash Tables

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

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

- Worst-case running time is $O(1)$
- Note that you may also need to search to check if it is already there

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)

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

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

- Worst-case running time is the same as the **search** operation

Analysis of Hashing with Chaining: Average Case

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

  Then search/delete operations may take $\Theta(n)$

- **Average case** depends on how well the hash function distributes the n keys among the m slots

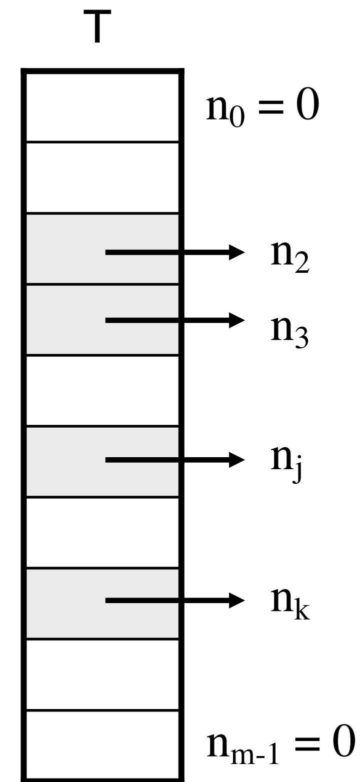
- **Simple uniform hashing**: Any given element is equally likely to hash into any of the m slots
  (i.e. when T is empty, collision probability Pr(h(x)=h(y)), is 1/m)

- Length of a list: n_j, $j = 0, 1, \ldots, m-1$
- 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 T: $\alpha = n/m$

  α = the average number of elements stored in a chain



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. Given any random x and y, probability of collision Pr(h(x)=h(y)), is 1/m)

Proof :

- need to search to the end of the list T[h(k)]
- Expected length of the list: $E[n_{h(k)}] = \alpha = n/m$
- Total time required : **O(1)** (to compute h(k)) + α → $\Theta(1+\alpha)$

Case II: Successful Search (i.e. item found in the table)

A successful search in a hash table takes expected time $\Theta(1 + \alpha/2)$ under the assumption of simple uniform hashing where $\alpha = n/m$ (expected length of the list)

If m (# of slots) is proportional to n (# of elements in the table):

$n = O(m)$ ➔ $\alpha = n/m = O(m)/m = O(1)$

➔ **Searching (both successful and unsuccessful) takes constant time O(1) on average**

Hash Functions

What makes a good hash function?

(1) Easy to compute
(2) Approximates a random function: considering all the keys, 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
- Minimize the chance that closely related keys hash to the same slot (i.e. 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
- Certain values of m are bad, e.g. power of 2 or non-prime numbers
- If $m = 2^p$ then $h(k)$ is just the least significant p bits of k
- Choose m to be a prime, not close to a power of 2

The Multiplication Method

$$h(k) = \lfloor m\,(\underbrace{k A - \lfloor k A \rfloor}_{\text{fractional part of kA}}) \rfloor \qquad 0 < A < 1$$

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

EXAMPLE: $m = 2^3$

.101101 (A)

110101 (k) **(kA) = 1001010.0110011**

Discard integer part: 1001010

Shift .0110011 by 3 bits to the left ➔ 011.0011

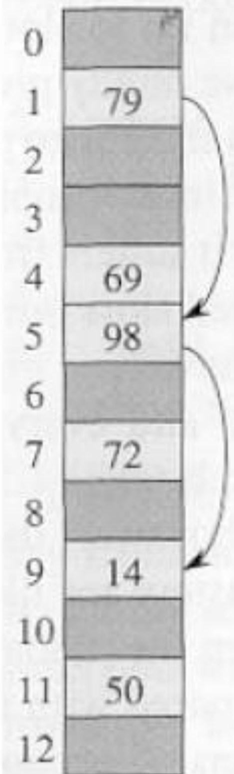
Take integer part: 011

Thus, $\boldsymbol{h(110101) = 011}$

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

e.g., insert 14



- **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 probe sequence!

Generalized hash function notation:

- A hash function contains two arguments now:

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

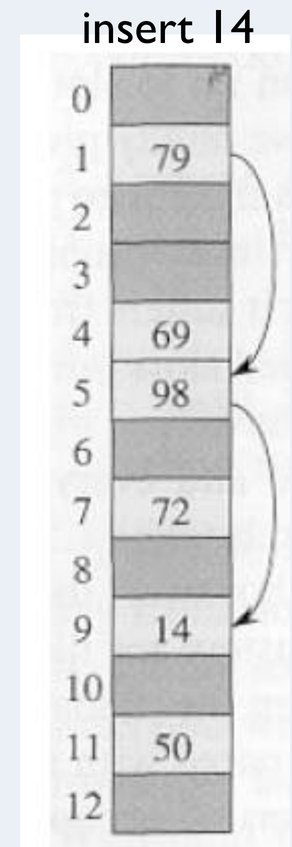
  h(k,p), p=0,1,...,m-1

- Probe sequences

  <h(k,0), h(k,1), ..., h(k,m-1)>

  - Must be a permutation of <0,1,...,m-1>
  - There are m! possible permutations
  - An optimum hash function has potential to produce all m! probe sequences

insert 14



Example: <1, 5, 9>

Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

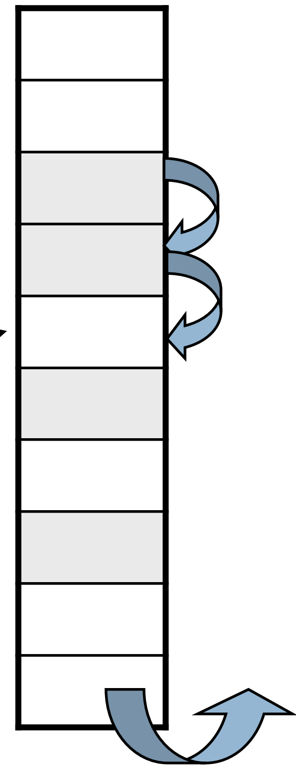
Linear probing: Inserting a key

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

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

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



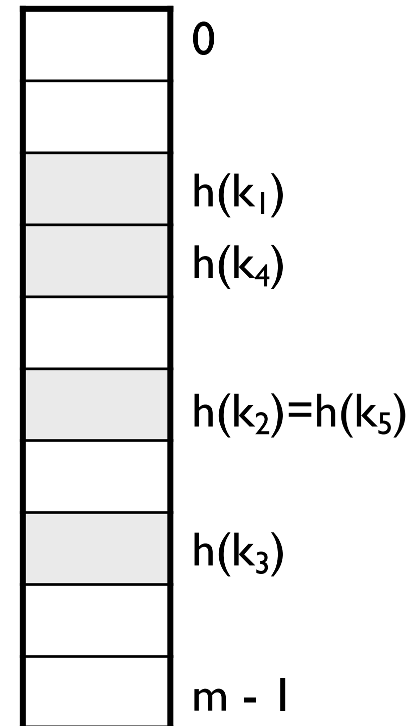
- Can generate m probe sequences maximum, why?

Linear probing: Searching for a key

- **Four cases:**
  1) Position in table is occupied with an element of equal key
  2) Position in table is marked as 'deleted'
  3) Position in table occupied with a different element
  4) Position in table is empty

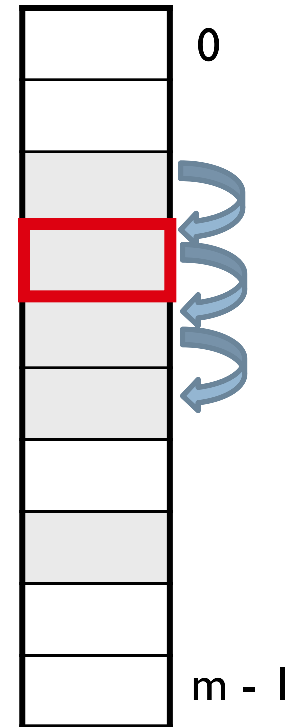
- **Cases 2 or 3:** probe the next higher index until the element is found or an empty position is found

- The process wraps around to the beginning of the table



Linear probing: Deleting a key

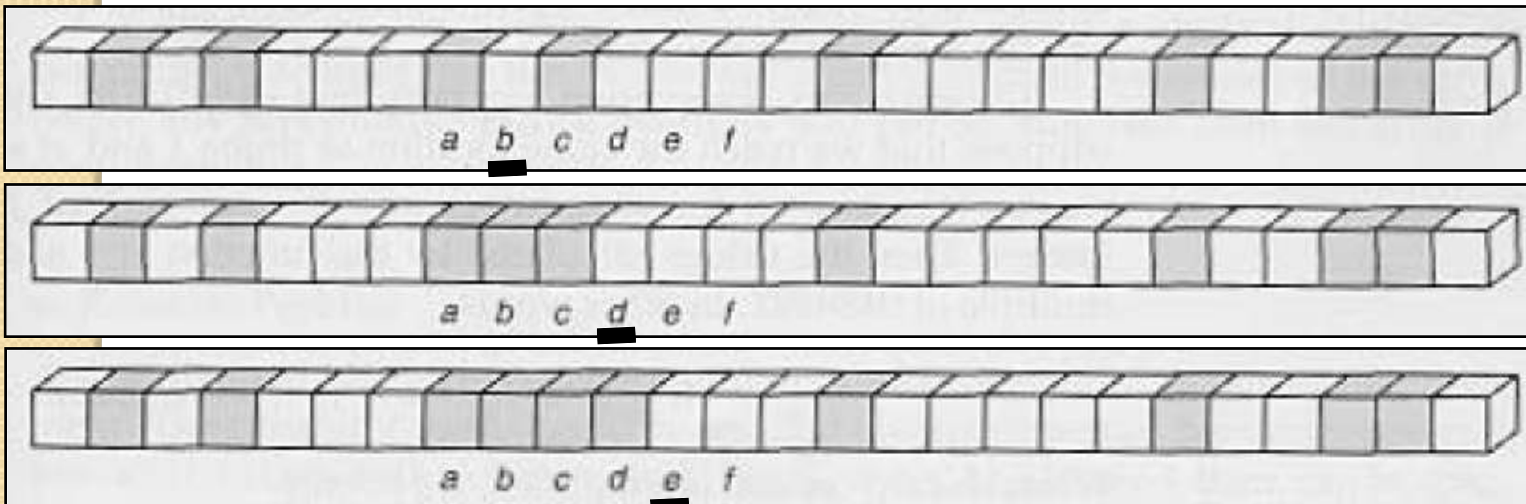
- Problems
  - Cannot mark the slot as empty
  - Otherwise, impossible to retrieve those keys inserted when that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion



Primary Clustering Problem

- Some slots become more likely than others
- Long chunks of occupied slots are created
  $\Rightarrow$ search time increases!

initially, all slots have probability 1/m



Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Quadratic probing

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \quad i = 0, 1, \ldots, m-1$

where h': $U \rightarrow (0,1,\ldots, m-1)$, $c_1, c_2 > 0$

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

$$h(k,i) = (h_1(k) + i\,h_2(k)) \ \mathbf{mod}\ m, \quad i=0,1,\ldots$$

- Initial probe: $h_1(k)$
- Second probe is offset by **$h_2(k)$ mod m**, so on ...

- Advantage: avoids clustering
- Disadvantage: harder to delete an element

- Can generate **m^2** probe sequences maximum. Why?
- (For two different keys k_a and k_b, even if $h_1(k_a)=h_1(k_b)$, it is unlikely that $h_2(k_a)=h_2(k_b)$, therefore two keys that initially probe to the same location may generate different probe sequences. Furthermore, after the initial probe, increment could be 1, or 2, or 3, or …m each one resulting in a unique probe sequence – a total of m. There are m initial probes each may in turn generate m sequences ➔ A total of m*m= **m^2** sequences)

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:**

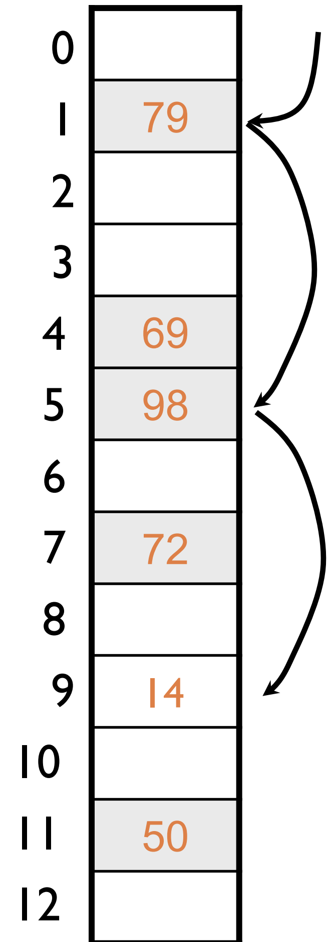
$h_1(14,0) = 14 \bmod 13 = 1$

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

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

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

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



Analysis of Open Addressing

Theorem 11.6: Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing

Proof: Ignore the problem of clustering and assume that all probe sequences are equally likely, then

Unsuccessful search analysis:

Probability(*probe hits an occupied cell*) = α (load factor $\alpha = n/m$)
Prob(*probe hits an empty cell*) = $1-\alpha$
probability that *a probe terminates in 2 steps* = $\alpha(1-\alpha)$
probability that *a probe terminates in k steps* = $\alpha^{k-1}(1-\alpha)$

What is the average number of steps in a probe?

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

Analysis of Open Addressing (cont'd)

Successful search:

$$E(\#steps) = \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

<u>Example</u>

Unsuccessful probe:

$\alpha = 0.5$ E(#steps) = 2

$\alpha = 0.9$ E(#steps) = 10

Successful probe:

$\alpha = 0.5$ E(#steps) = 1.386

$\alpha = 0.9$ E(#steps) = 2.558