Collision

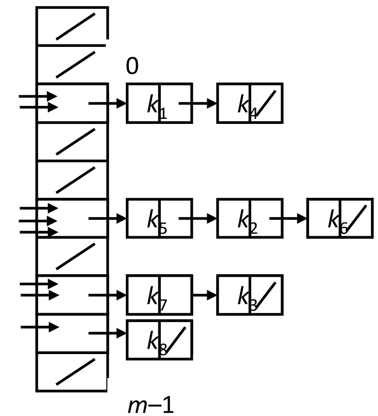
- Two or more keys hash to the same slot.
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely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

Collision Resolution Techniques

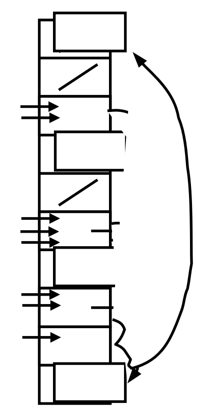
- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Collision Resolution Techniques

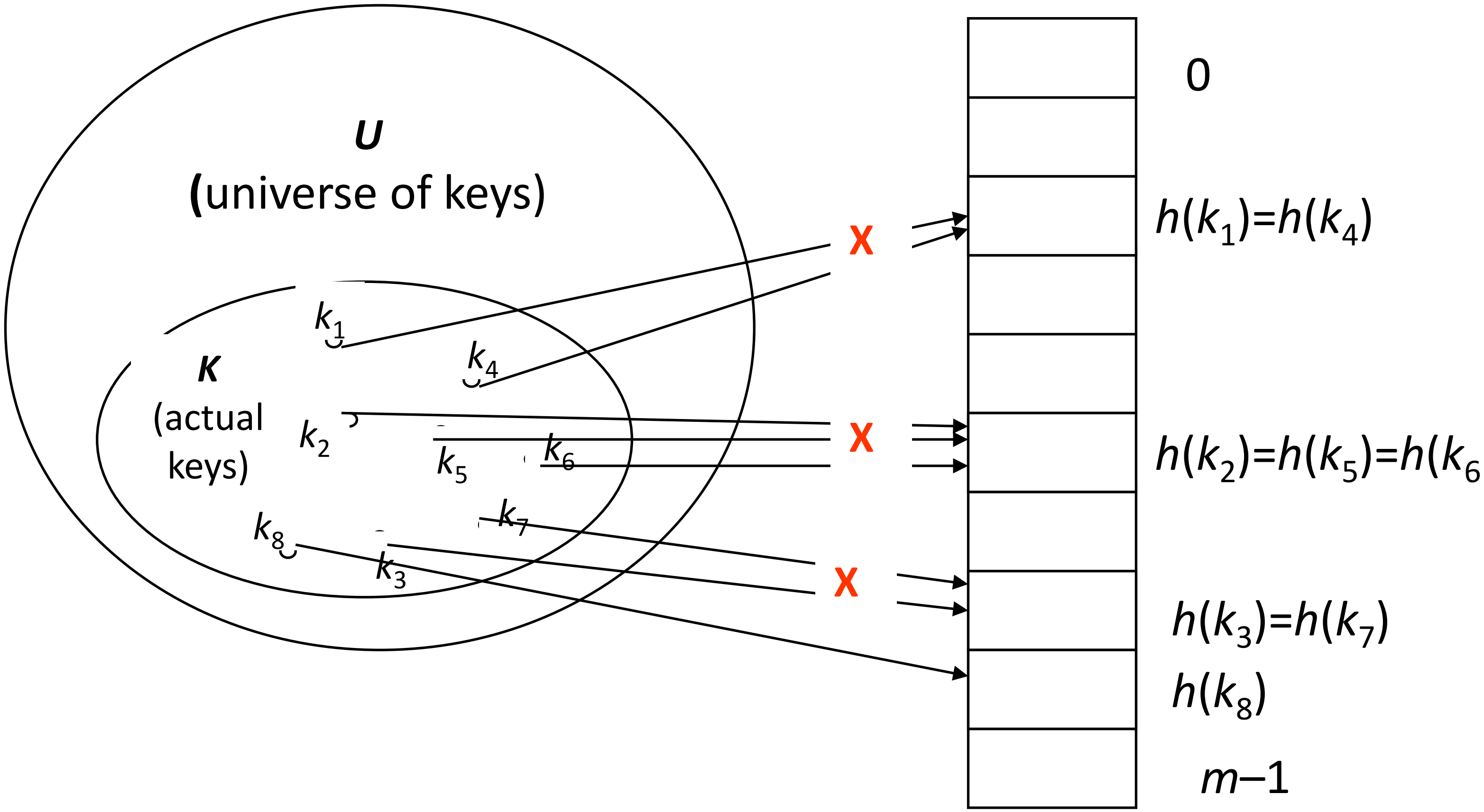
- <u>Chaining</u>:
    - Store all elements that hash to the same slot in a linked list.
    - Store a pointer to the head of the linked list in the hash table slot.



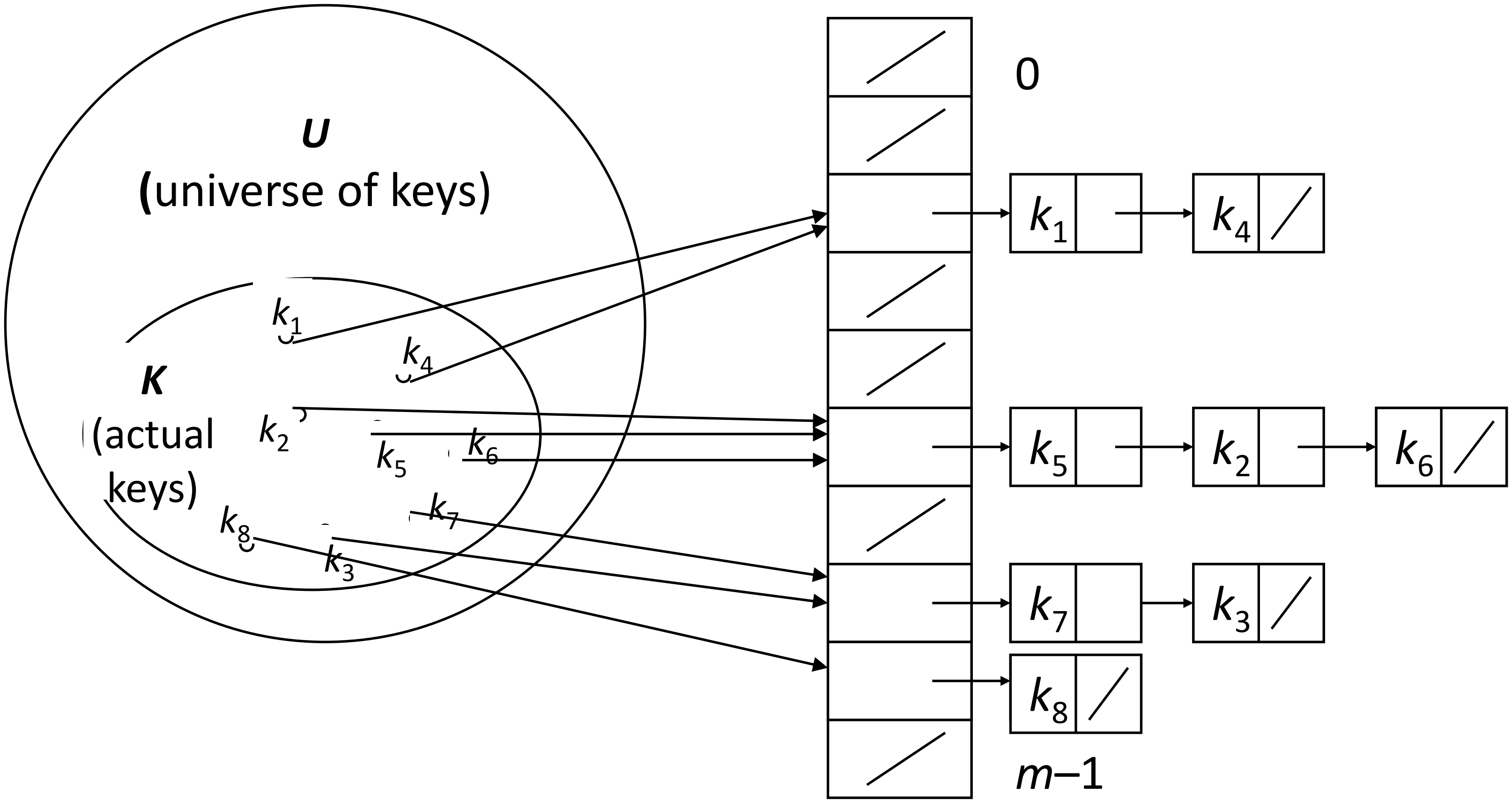
- <u>Open Addressing</u>:
    - All elements stored in hash table itself.
    - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining



Collision Resolution by Chaining



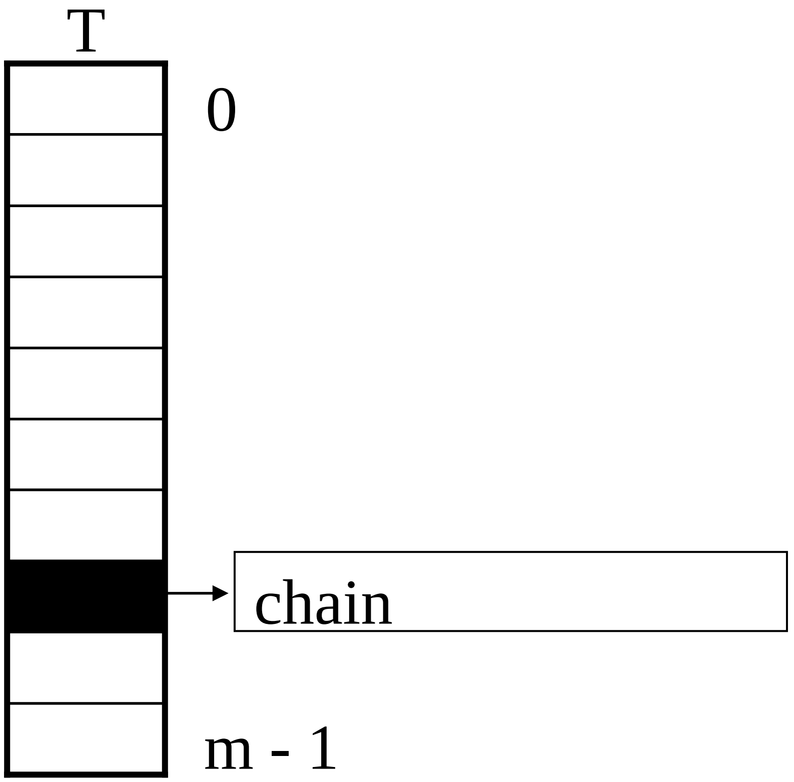
Hashing with Chaining

Dictionary Operation

- Chained-Hash-Insert (*T, x*)
  - Insert *x* at the head of list *T*[*h*(*key*[*x*])].
  - Worst-case complexity : *O*(1).
- Chained-Hash-Delete (*T, x*)
  - Delete *x* from the list *T*[*h*(*key*[*x*])].
  - Worst-case complexity : proportional to length of list with singly-linked lists. *O*(1) with doubly-linked lists.
- Chained-Hash-Search (*T, k*)
  - Search an element with key *k* in list *T*[*h*(*k*)].
  - Worst-case complexity : proportional to length of list.

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

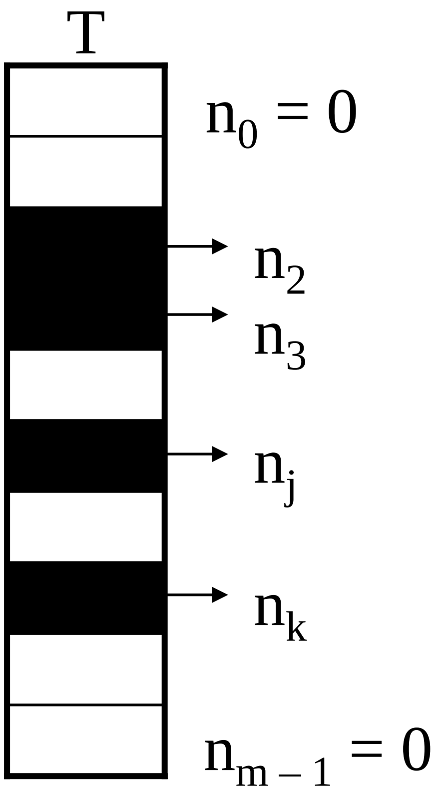
•**Simple uniform hashing** assumption:
Any given element is equally likely to hash into any of the m slots (i.e., probability of collision Pr(h(x)=h(y)), is 1/m)

•Length of a list:

$T[j] = n_j, \quad j = 0, 1, . . . , m – 1$

•Number of keys in the table: $n = n_0 + n_1 + \cdot \cdot \cdot + n_{m-1}$

•Average value of n_j: $E[n_j] = \alpha = n/m$

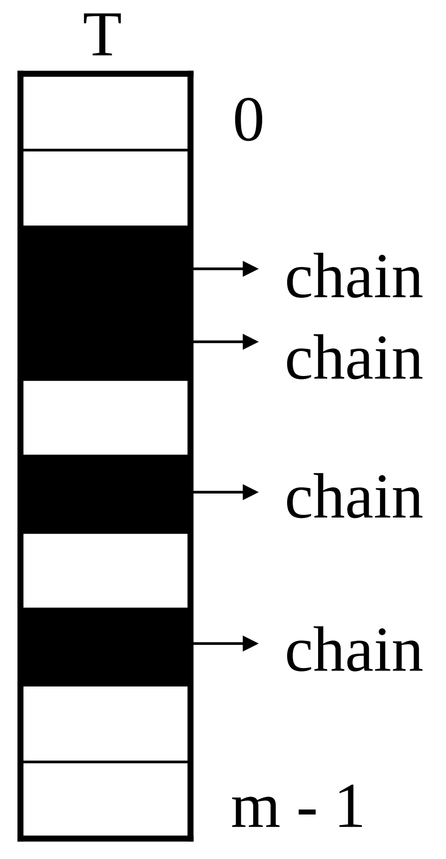


Load Factor of a Hash Table

- Load factor of a hash table T:

  $\alpha = n/m$

  - n = # of elements stored in the table
  - m = # of slots in the table
- 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 : $T[h(k)]$
- 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:
  $O(1)$ (for computing the hash function) + $\alpha \rightarrow \Theta(1+\alpha)$

Case 2: Successful Search

Theorem: A successful search takes expected time $\Theta(1+\alpha)$.

Proof :

- Let x_i be the i^{th} element inserted into the table, and let $k_i = key[x_i]$.
- Define indicator random variables $X_{ij} = I\{h(k_i) = h(k_j)\}$, for all i, j.
- Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m$

  $\Rightarrow E[X_{ij}] = 1/m$.
- Expected number of elements examined in a successful search is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

Case 2: Successful Search

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

(linearity of expectation)

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$=1+\frac{n-1}{2m}$$

$$=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Expected total time for a successful search
= Time to compute hash function + Time to search
= $\boldsymbol{O(1+1+\alpha/2 - \alpha/2n) = O(1+\alpha)}$.

Analysis of Search in Hash Tables

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.

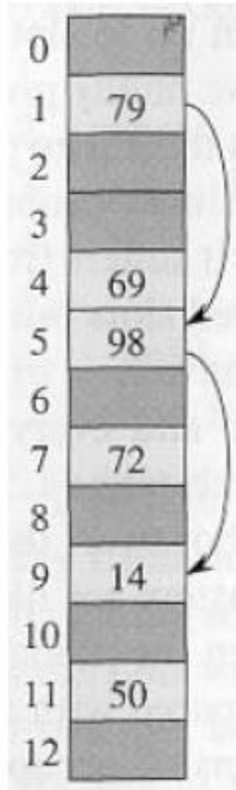
$\Rightarrow$ *Searching takes constant time on average.*

- Insertion is $O(1)$ in the worst case.
- Deletion takes $O(1)$ worst-case time when lists are doubly linked.
- Hence, ***<u>all dictionary operations take O(1) time on average with hash tables with chaining.</u>***

Open Addressing

- If we have enough contiguous memory to store all the keys (m > N) $\Rightarrow$ 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 ...
- Search time depends on the length of the probe sequence!

e.g., insert 14



Generalize hash function notation:

- A hash function contains two arguments now:

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

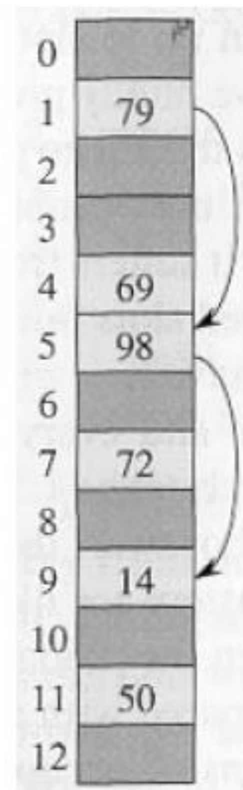
  h(k,p), p=0,1,...,m-1

- 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
  - Good hash functions should be able to produce all *m!* probe sequences

insert 14



Probe sequence: <1, 5, 9>

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)

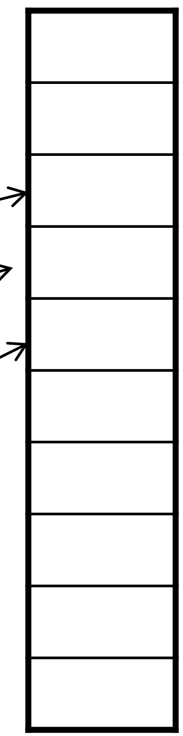
$$h(k,i) = (h_1(k) + i) \bmod m$$

$$i=0,1,2,\ldots$$

First slot probed: $h_1(k)$

Second slot probed: $h_1(k) + 1$

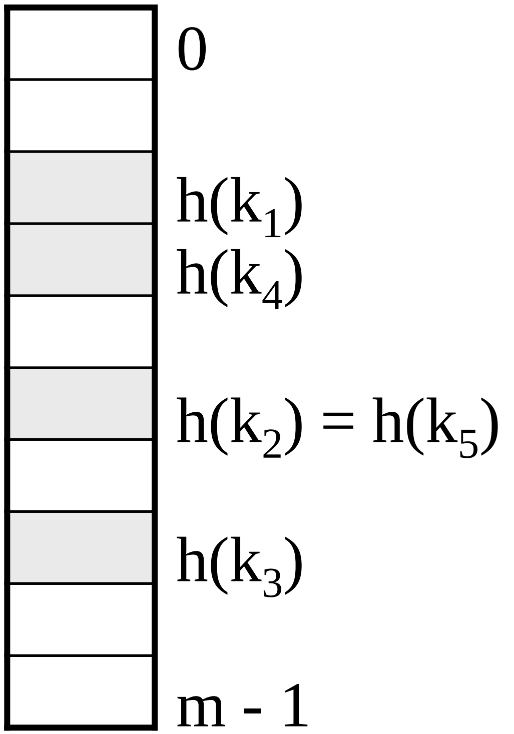
Third slot probed: $h_1(k)+2$, and so on



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

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 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

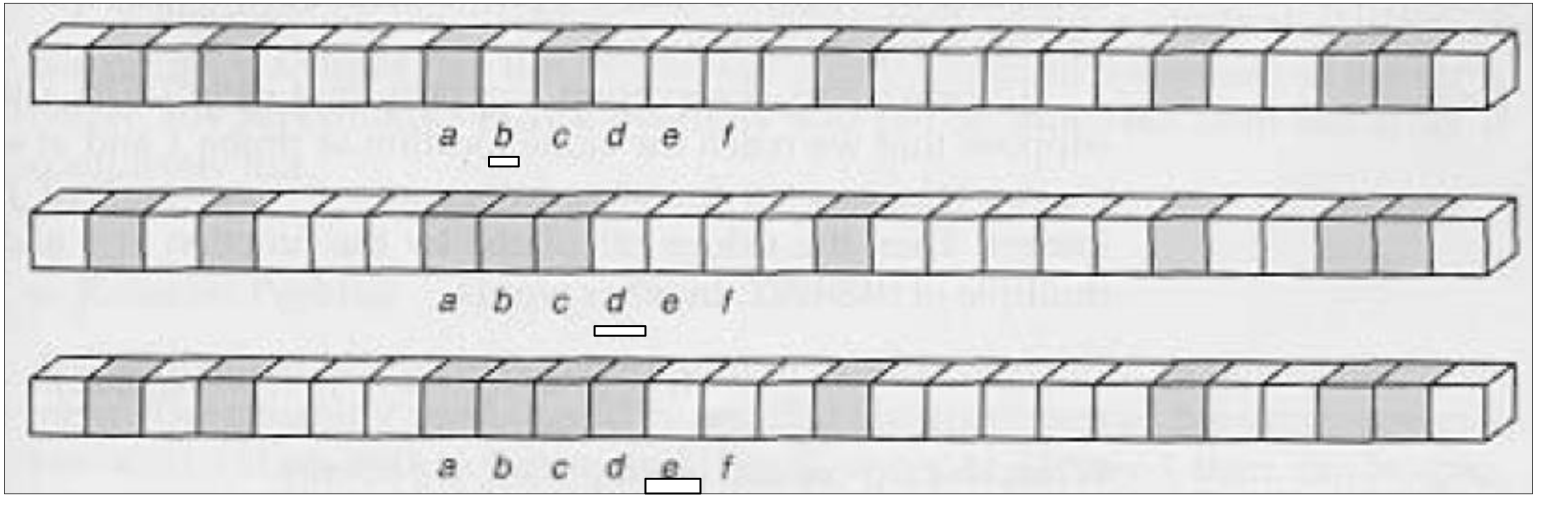
- Problems
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion
- Searching will be able to find all the keys

Primary Clustering Problem

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

$\Rightarrow$ <u>average insert & search time increases!!</u>

initially, all slots have probability 1/m



Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Quadratic probing

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

  key Probe number Auxiliary hash function
- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number i.
- Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same.

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 $h_2(k)$ mod m, so on ...
- Advantage: avoids clustering
- Disadvantage: harder to delete an element

Double Hashing: Example

$h_1(k) = k \bmod 13$

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

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

- Insert key 14:

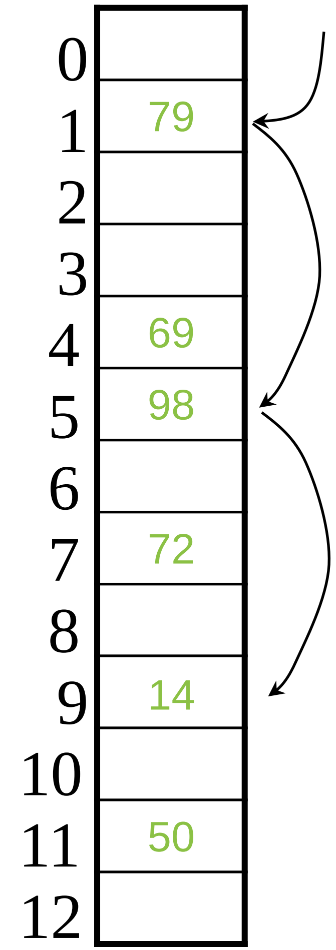
$h_1(14,0) = 14 \bmod 13 = 1$

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

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

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

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



Analysis of Open Addressing

- Analysis is in terms of load factor α.
- **Assumptions:**
  - Assume that the table never completely fills, so $n < m$ and $\alpha < 1$.
  - Assume uniform hashing.
  - <u>No</u> deletion.
  - All probe sequences are equally likely

Analysis of Open Addressing

- **Unsuccessful retrieval**:

  Prob(probe hits an occupied cell) = α

  Prob(probe hits an empty cell) = 1- α

  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?

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

Analysis of Open Addressing

- **successful retrieval**:
  The expected number of probes in a successful search in an open-address hash table is at most $(1/\alpha) \log (1/(1-\alpha))$.

Unsuccessful retrieval:

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

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

Successful retrieval:

$\alpha = 0.5$ E(#steps) = 3.387

$\alpha = 0.9$ E(#steps) = 3.670