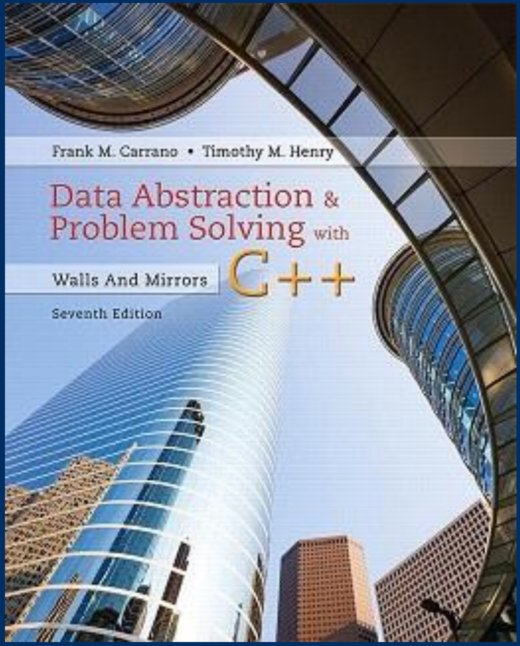
Hashing

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

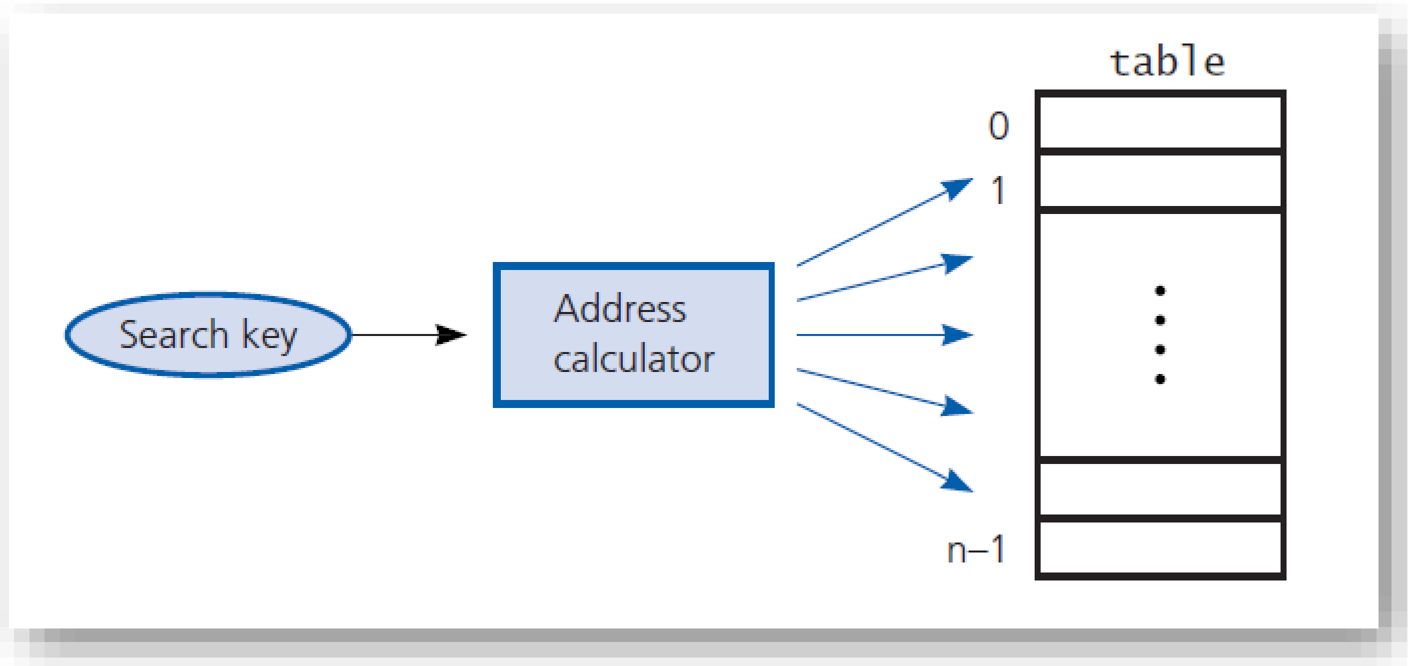
Reminders

- Assignment 6 is available
  - Due Monday April 23rd at 2pm
- TA
  - Shehryar Khattak,
    Email: *shehryar [at] nevada {dot} unr {dot} edu*,
    Office Hours: Friday, 11:00 am - 1:00 pm at ARF 116

Hashing as a Dictionary Implementation

- Situations occur for which search-tree implementations are not adequate.
- Consider a method which acts as an “address calculator” which determines an array index
  - Used for add, getValue, remove operations
- Called a hash function
  - Tells where to place item in a hash table

Hashing as a Dictionary Implementation



- Address calculator

Hashing

- is a means used to order and access elements in a list quickly by using a function of the key value to identify its location in the list.
  - the goal is O(1) time
- The function of the key value is called a hash function

Hashing

Idea:

– Use a function **h** to compute the slot for each key

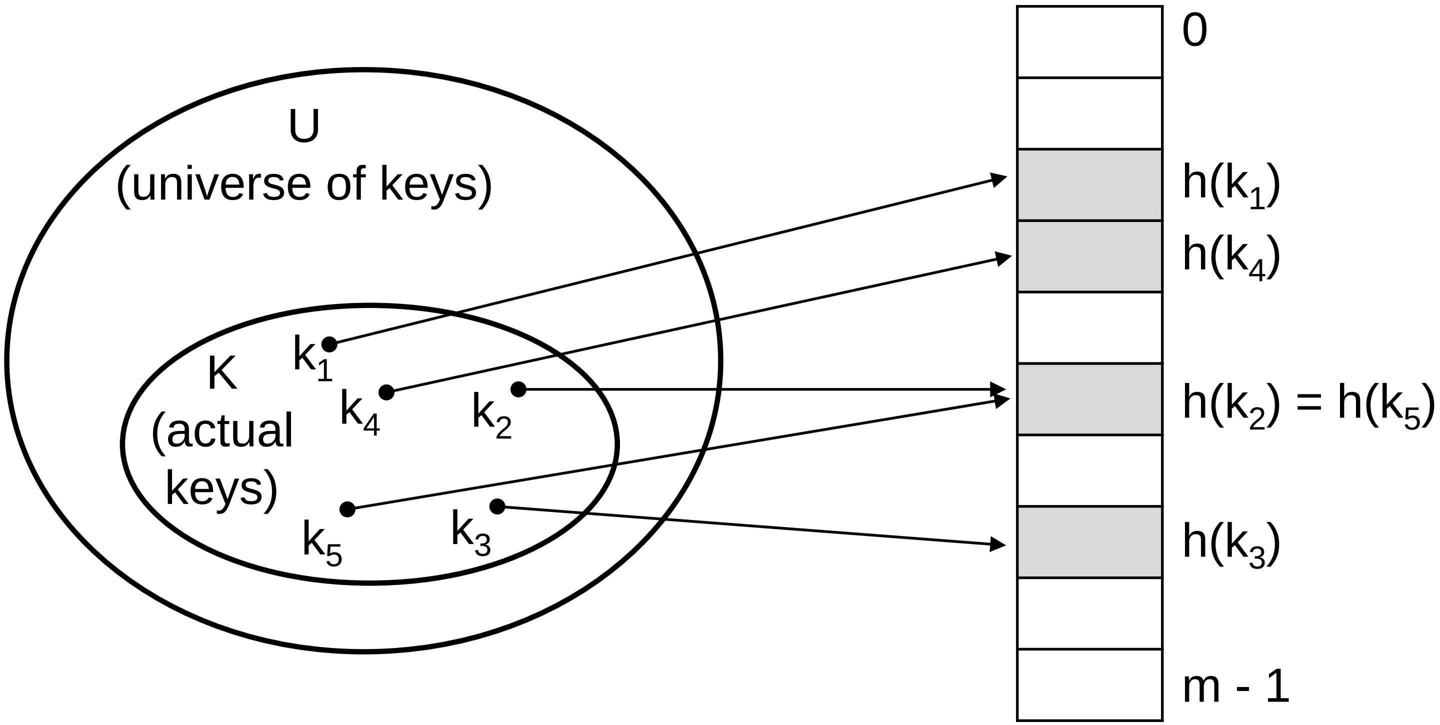
– Store the element in slot **h(k)**

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

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

- We say that k **hashes** to slot h(k)

Hashing (cont'd)



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

hash table size: **m**

Hashing (cont'd)

Example 2: Suppose that the keys are 9-digit social security numbers (SSN)

Possible hash function

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

$$\text{e.g., if } ssn = 10123411 \text{ then } h(10123411) = 11)$$

Advantages of Hashing

- Reduce the range of array indices handled:

  m instead of $|U|$

  where m is the hash table size: T[0, …, m-1]

- Storage is reduced.

- $O(1)$ search time (i.e., under assumptions).

Properties of Good Hash Functions

- **Good hash function properties**

(1)Easy to compute

(2) Approximates a random function

i.e., for every input, every output is equally likely.

(3) Minimizes the chance that similar keys hash to the same slot

i.e., strings such as pt and pts should hash to different slot.

Hashing as a Dictionary Implementation

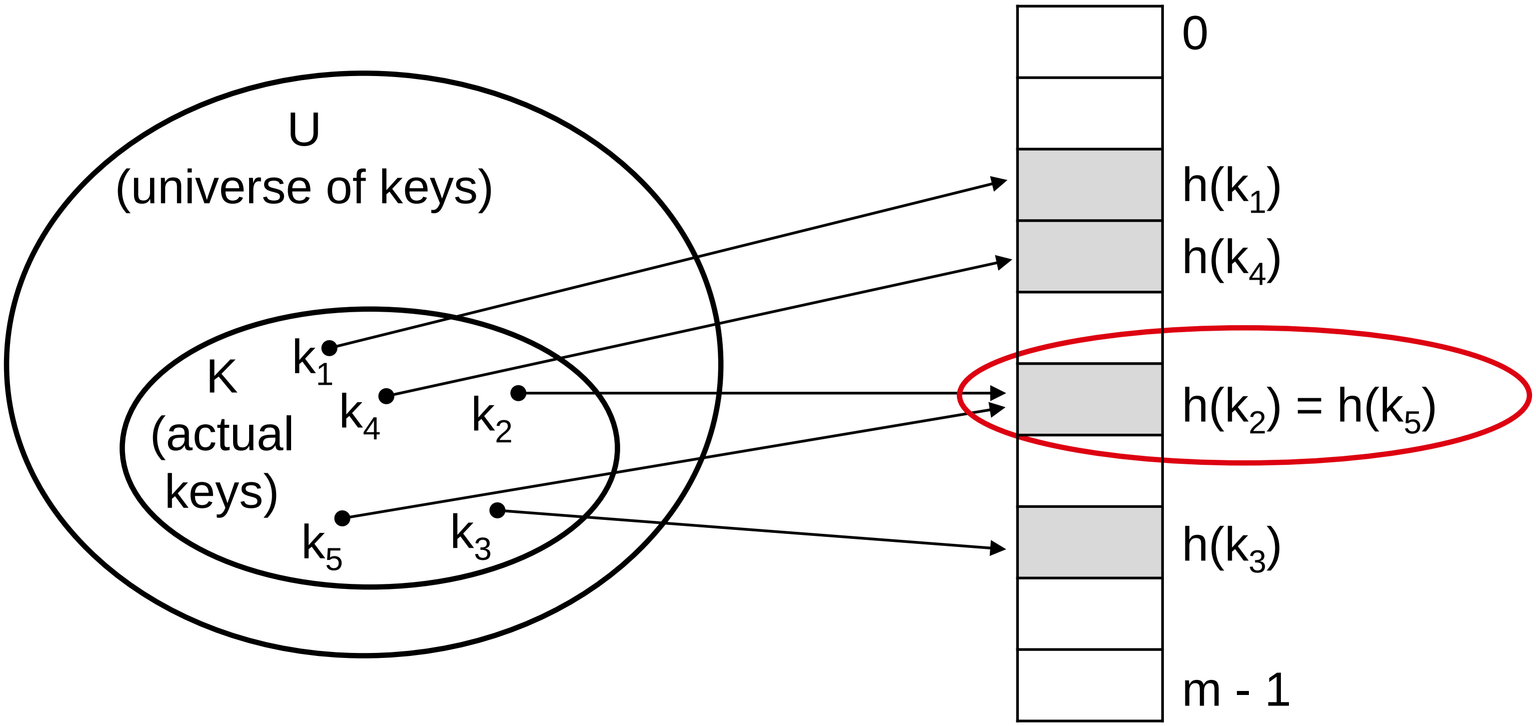
- Perfect hash function
  - Maps each search key into a unique location of the hash table
  - Possible if you know all the search keys
- Collision occurs when hash function maps more than one entry into same array location
- Hash function should
  - Be easy, fast to compute
  - Place entries evenly throughout hash table

Hash Functions

- Sufficient for hash functions to operate on integers – examples:
  - Select digits from an ID number
  - Folding – add digits, sum is the table location
  - Modulo arithmetic $h(x) = x \bmod tableSize$
  - Convert character string to an integer – use ASCII values

Collisions

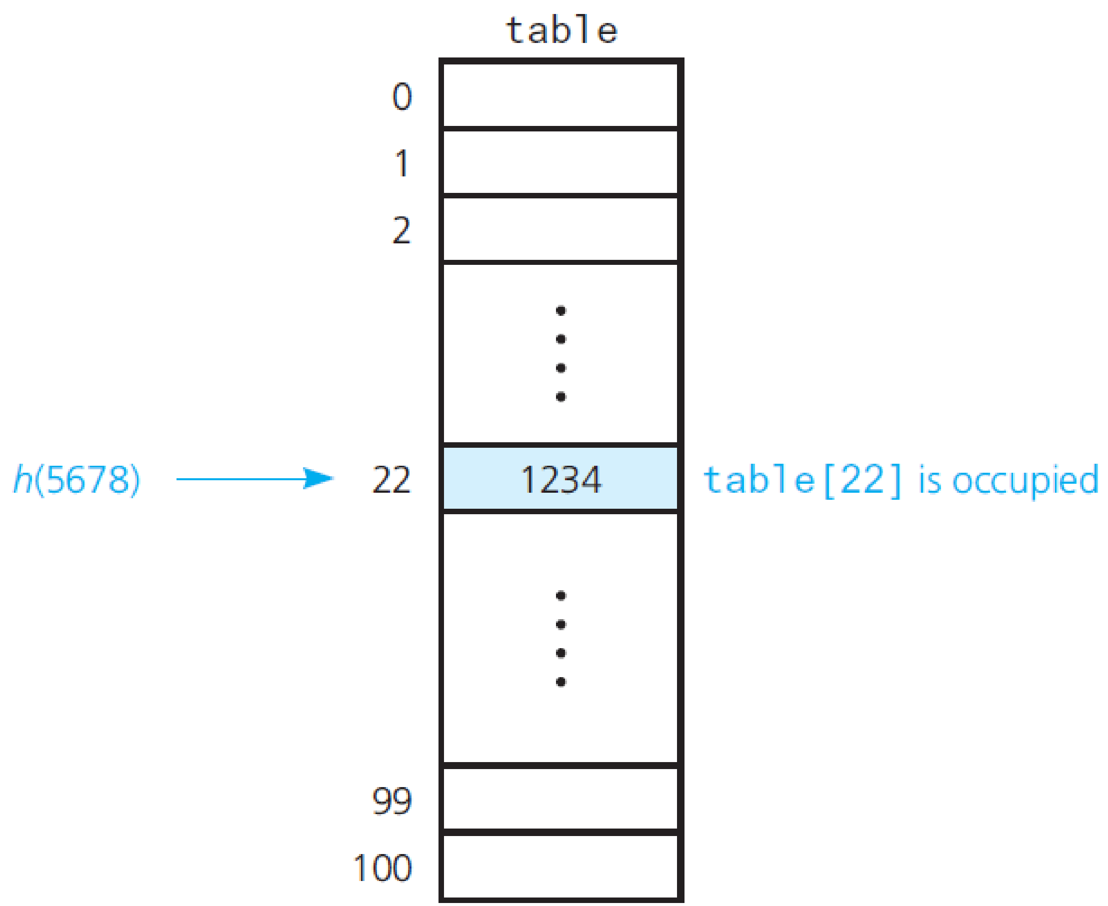
Collisions occur when $h(k_i)=h(k_j)$, $i \neq j$



Collisions (cont'd)

- 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 might not be easy.

Resolving Collisions with Open Addressing



- A collision

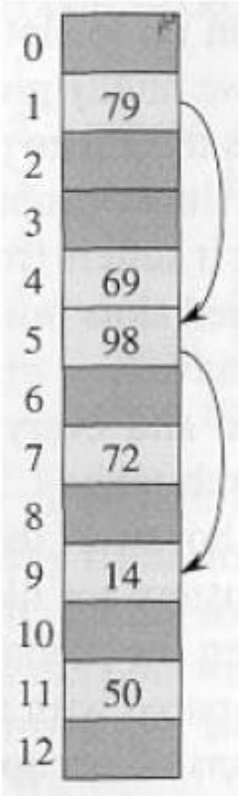
Resolving Collisions with Open Addressing

- Approach 1: Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
  - Increase size of hash table

Open Addressing

- Idea: 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 probe sequence.
  - Deletion: need to be careful!
- Search time depends on the length of probe sequences!

e.g., insert 14



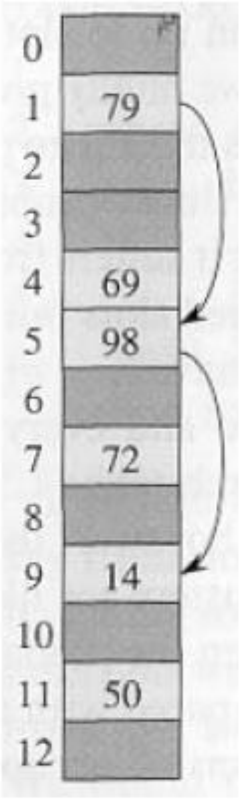
probe sequence: <1, 5, 9>

Generalize hash function notation:

- A hash function contains two arguments now:

  (i) key value, and (ii) probe number

e.g., insert 14



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

- Probe sequence:

  $<h(k,0), h(k,1), h(k,2), \ldots >$

- Example:

Probe sequence: <h(14,0), h(14,1), h(14,2)>=<1, 5, 9>

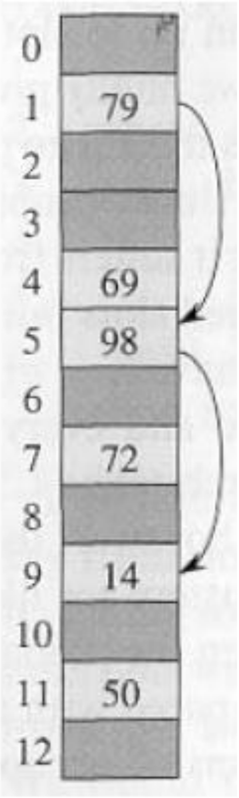
Generalize hash function notation:

– Probe sequence must be a permutation of

<0,1,...,m-1>

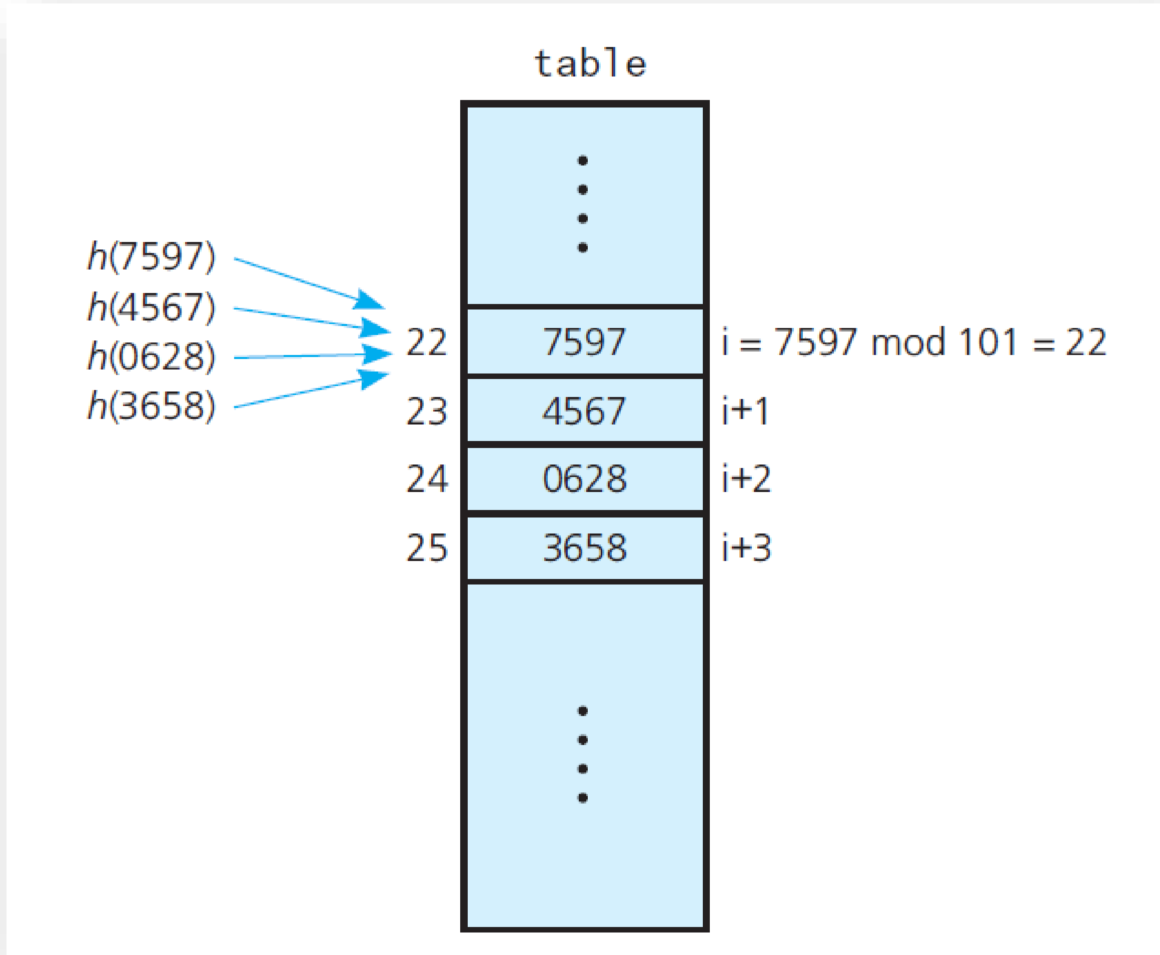
– There are m! possible permutations

e.g., insert 14



Probe sequence: <h(14,0), h(14,1), h(14,2)>=<1, 5, 9>

Resolving Collisions with Open Addressing



- Linear probing with h(x) = x mod 101

Linear probing: Inserting a key

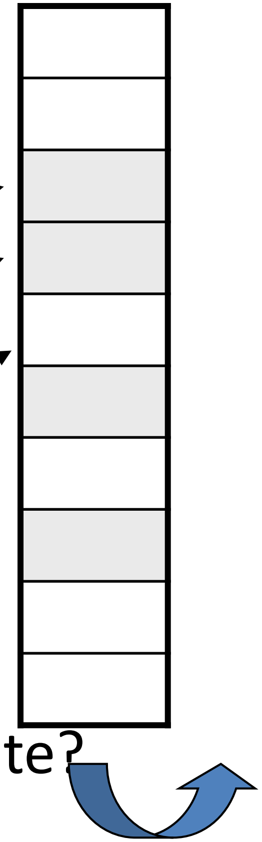
- **Idea:** when there is a collision, check the next available position in the table:

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

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

- $i=0$: first slot probed: $h_1(k)$
- $i=1$: second slot probed: $h_1(k) + 1$
- $i=2$: third slot probed: $h_1(k)+2$, and so on

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



- How many probe sequences can linear probing generate?

  m probe sequences maximum

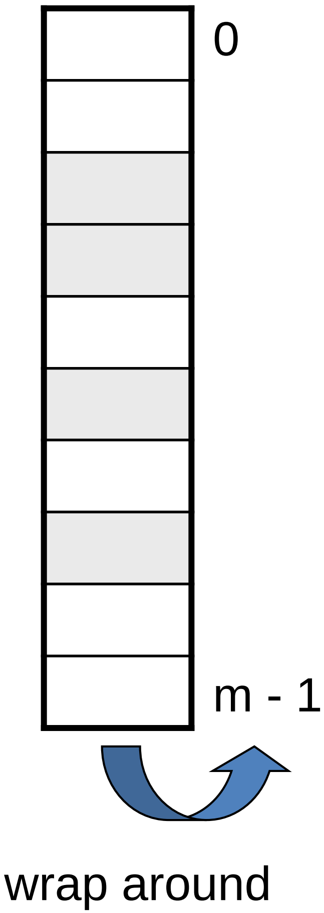
Linear probing: Searching for a key

- Given a key, generate a probe sequence using the same procedure.
- Three cases:

  (1) Position in table is occupied with an element of equal key→ FOUND

  (2) Position in table occupied with a different element → KEEP SEARCHING

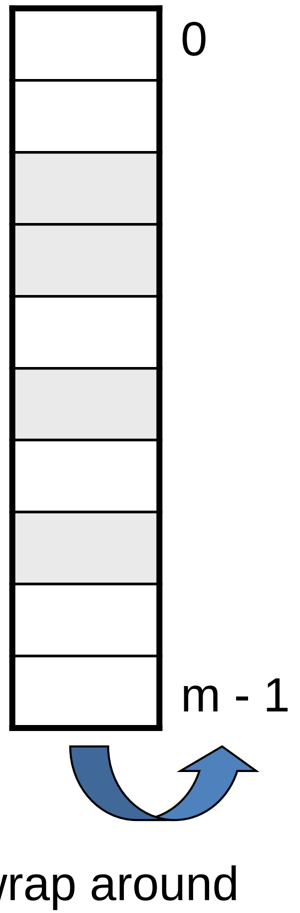
  (3) Position in table is empty→ NOT FOUND



Linear probing: Searching for a key

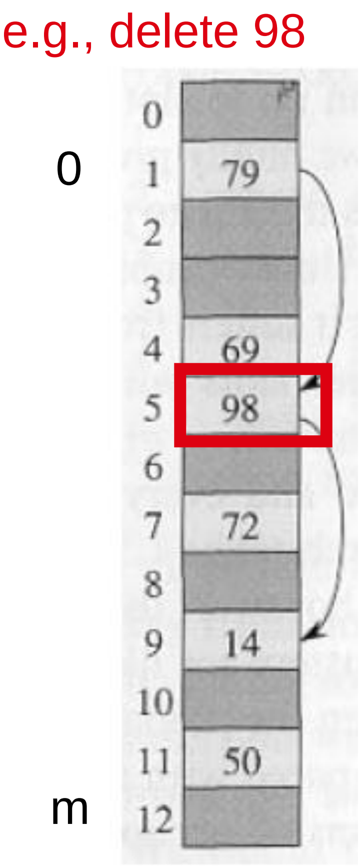
- Running time depends on the length of the probe sequences.

- Need to keep probe sequences short to ensure fast search.



Linear probing: Deleting a key

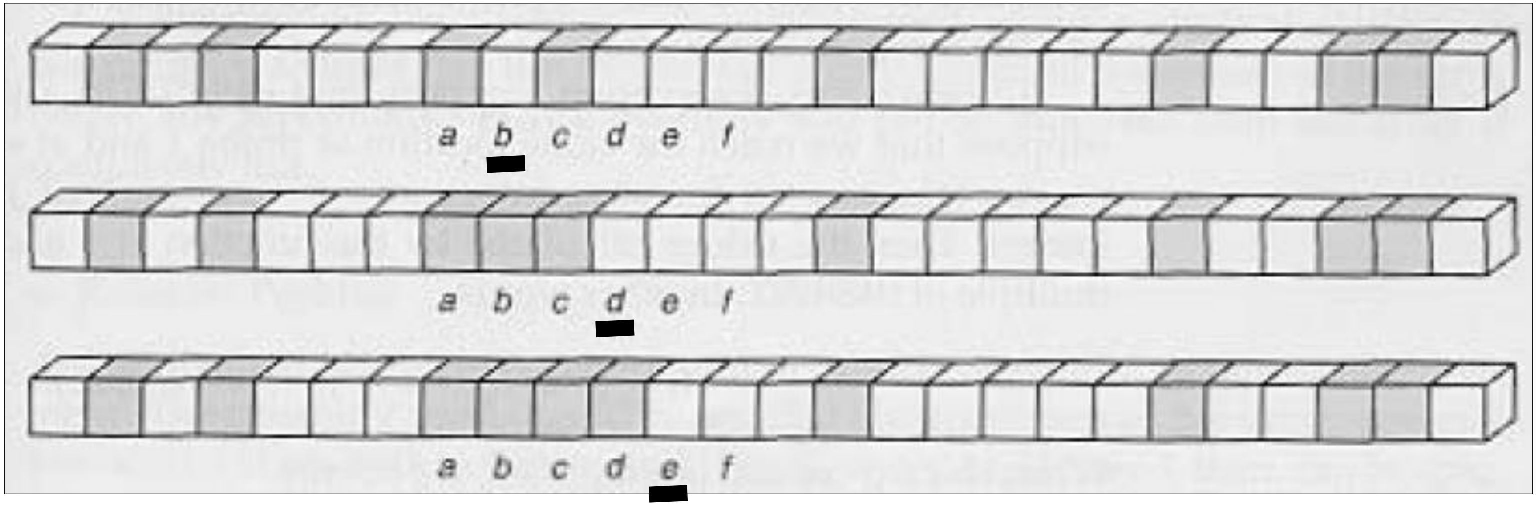
- First, find the slot containing the key to be deleted.
- Can we just mark the slot as empty?
  - It would be impossible to retrieve keys inserted after that slot was occupied!
- Solution
  - "Mark" the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion.



Primary Clustering Problem

- Long chunks of occupied slots are created.
- As a result, some slots become more likely than others.
- Probe sequences increase in length. $\Rightarrow$ search time increases!!

initially, all slots have probability 1/m

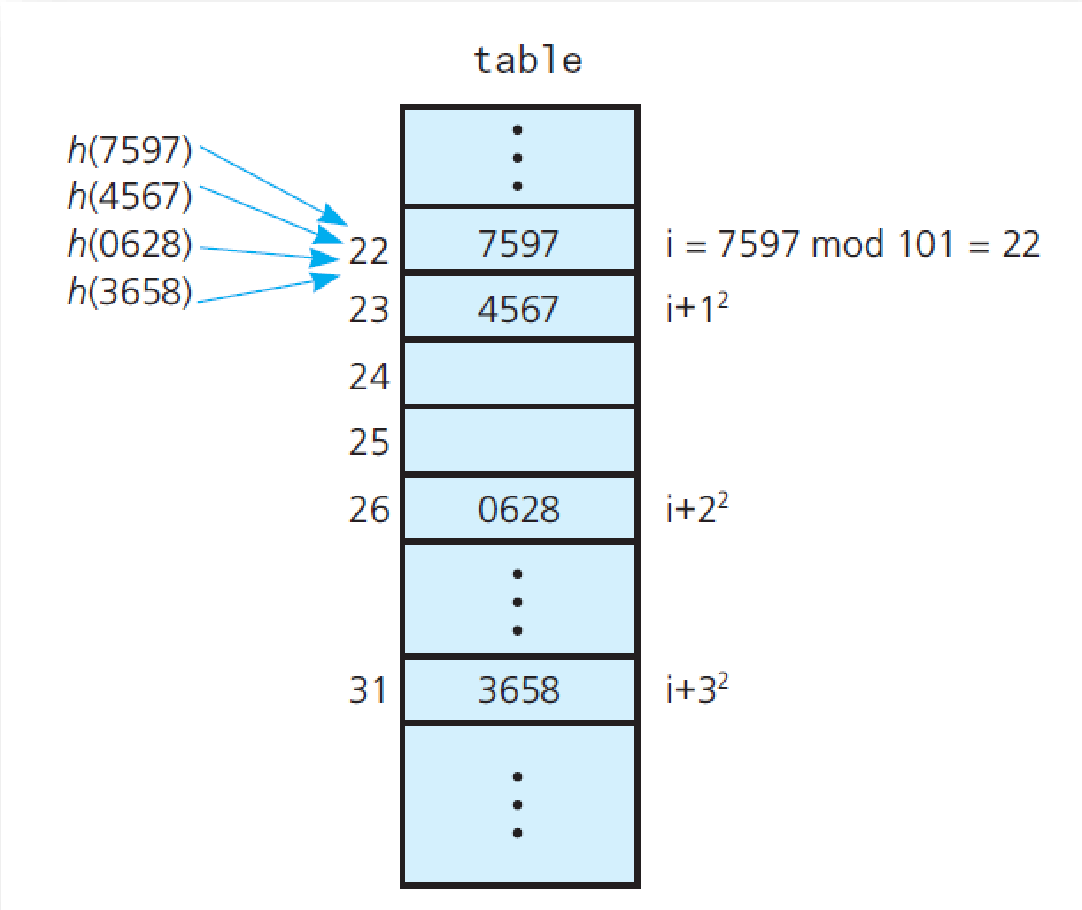


Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Resolving Collisions with Open Addressing



- Quadratic probing with h(x) = x mod 101

Quadratic probing

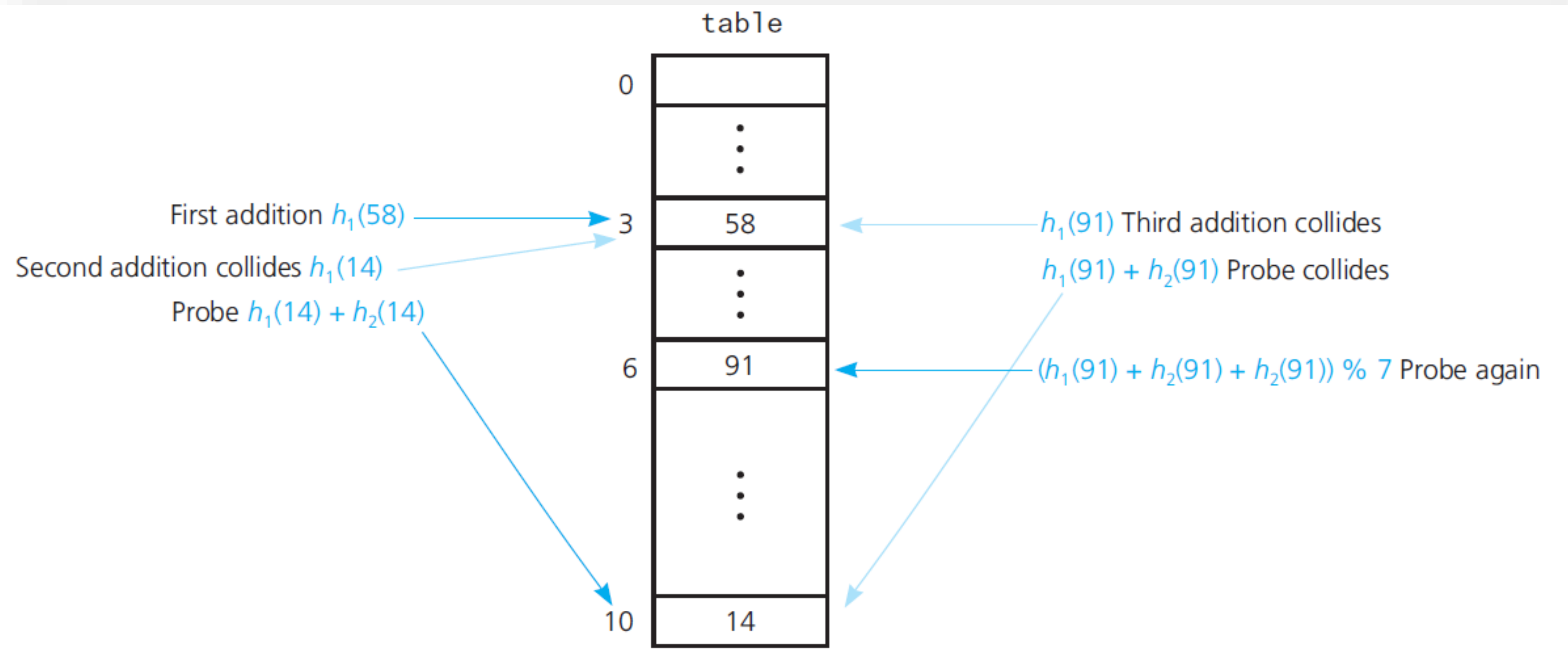
$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \text{ where } h': U --> (0,1,...,m-1)$$

i=0,1,2,...

- Clustering is less serious but still a problem
  - *secondary clustering*
- How many probe sequences can quadratic probing generate?

m -- the initial position determines probe sequence

Resolving Collisions with Open Addressing



- Double hashing during the addition of 58, 14, and 91

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 $h_2(k) \bmod m$, so on ...
- Advantage: handles clustering better
- Disadvantage: more time consuming
- How many probe sequences can double hashing generate?

m^2

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:

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

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

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

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

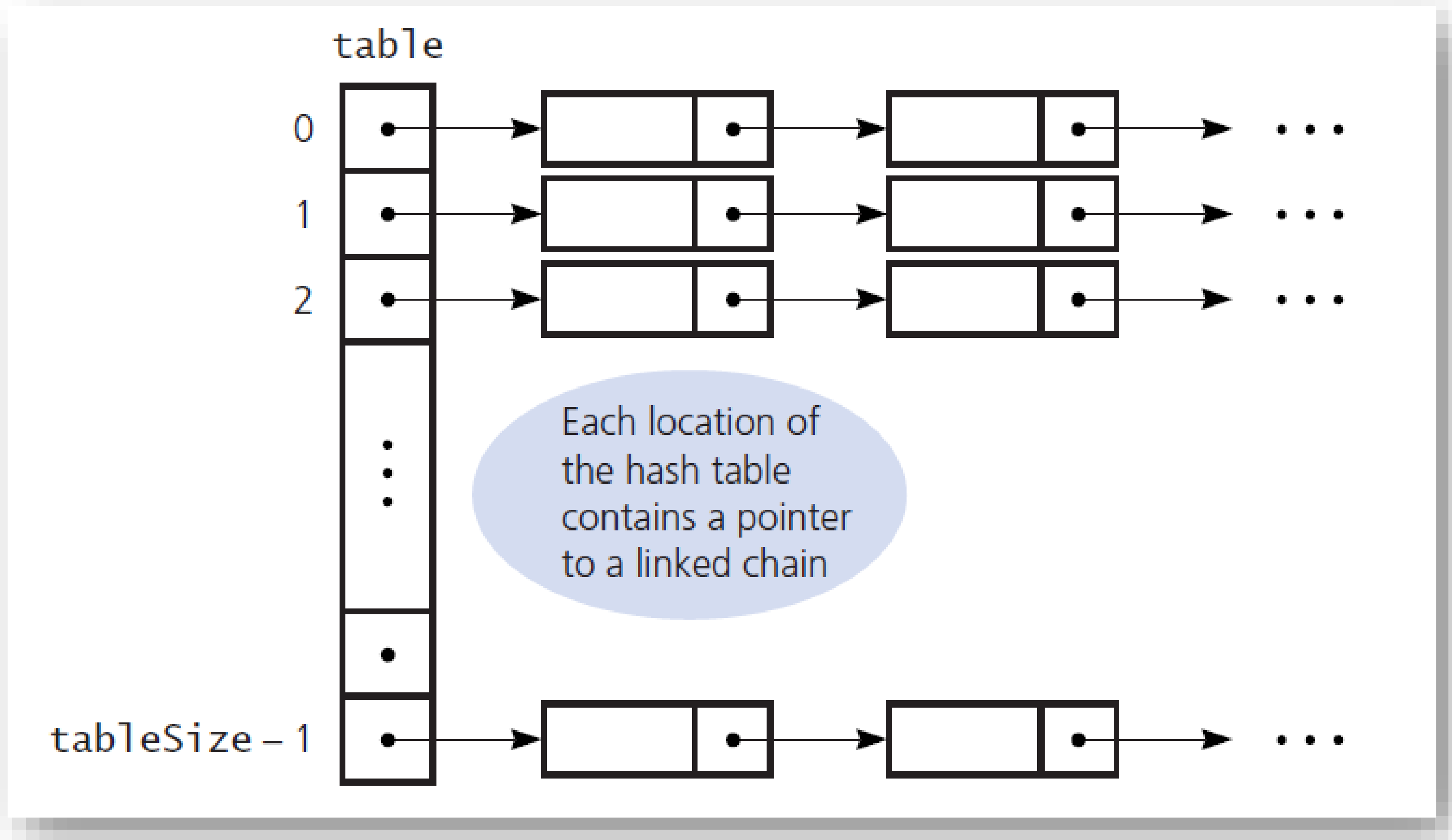
$= (1 + 8) \bmod 13 = 9$

0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Resolving Collisions with Open Addressing

- Approach 2: Resolving collisions by restructuring the hash table
  - Buckets
  - Separate chaining

Resolving Collisions with Open Addressing



- Separate chaining

Chaining

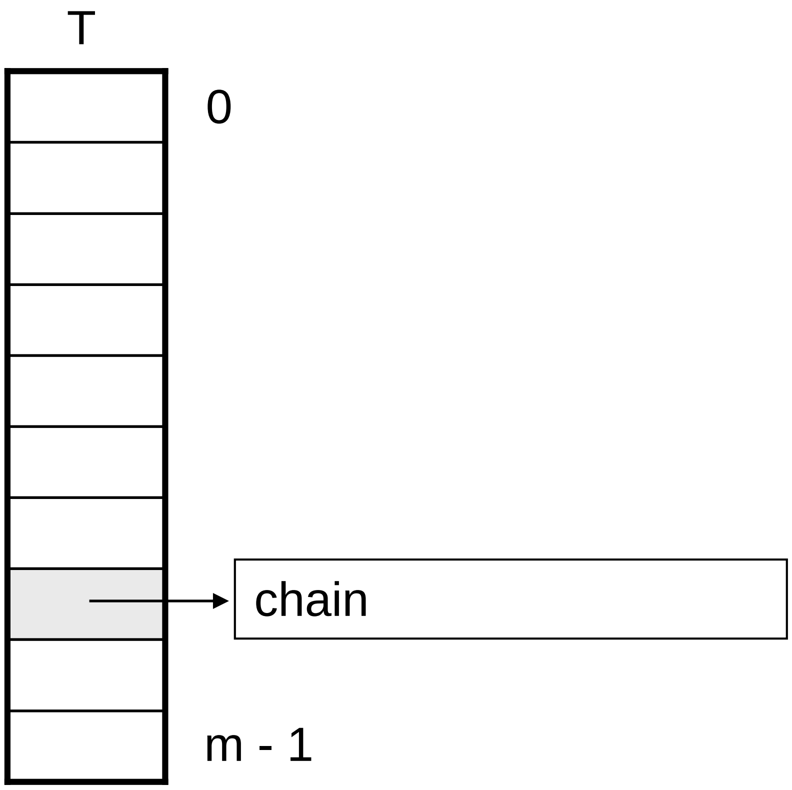
- How to choose the size of the hash table **m**?
  - Small enough to avoid wasting space
  - Large enough to avoid many collisions and keep linked-lists short
  - Typically 1/5 or 1/10 of the total number of elements
- Should we use sorted or unsorted linked lists?
  - Unsorted
    - Insert is fast
    - Can easily remove the most recently inserted elements

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

then $O(n)$ plus time to compute the hash function



Analysis of Hashing with Chaining: Average Case

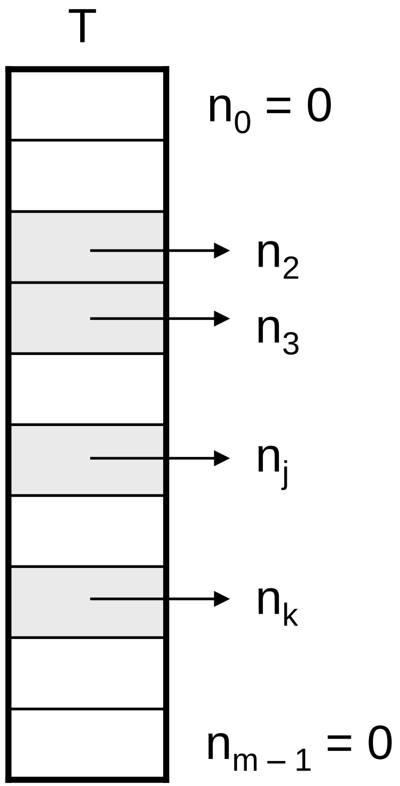
- It depends on how well the hash function distributes the n keys among the m slots
- Under the following assumptions:

  (1) **n = O(m)**

  (2) any given element is **equally likely** to hash into any of the m slots

  i.e., simple uniform hashing property

then → O(1) time plus time to compute the hash function



The Efficiency of Hashing

- Load factor measures how full a hash table is

$$\alpha = \frac{\text{Current number of table entries}}{\text{tableSize}}$$

- Unsuccessful searches
  - Generally require more time than successful
- Do not let the hash table get too full

The Efficiency of Hashing

- Linear probing – average number of comparisons

$$\frac{1}{2}\left[1 + \frac{1}{1 - \alpha}\right]$$ for a successful search, and

$$\frac{1}{2}\left[1 + \frac{1}{(1 - \alpha)^2}\right]$$ for an unsuccessful search

The Efficiency of Hashing

- Quadratic probing and double hashing – average number of comparisons

$$\frac{-\log_e(1-\alpha)}{\alpha} \quad \text{for a successful search,}$$

$$\frac{1}{1-\alpha} \quad \text{for an unsuccessful search}$$

The Efficiency of Hashing

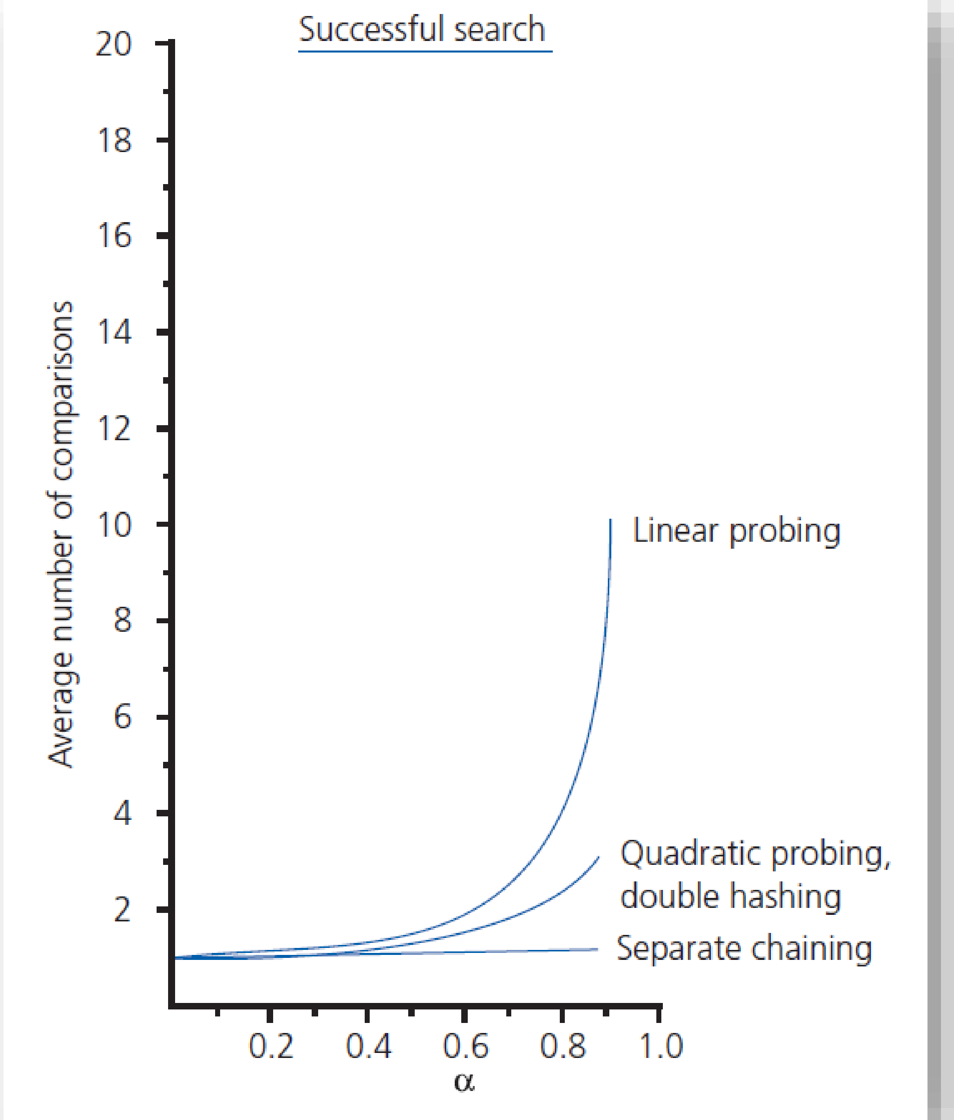
- Efficiency of the retrieval and removal operations under the separate-chaining approach

$$1 + \frac{\alpha}{2} \qquad \text{for a successful search,}$$

$$\alpha \qquad \text{for an unsuccessful search}$$

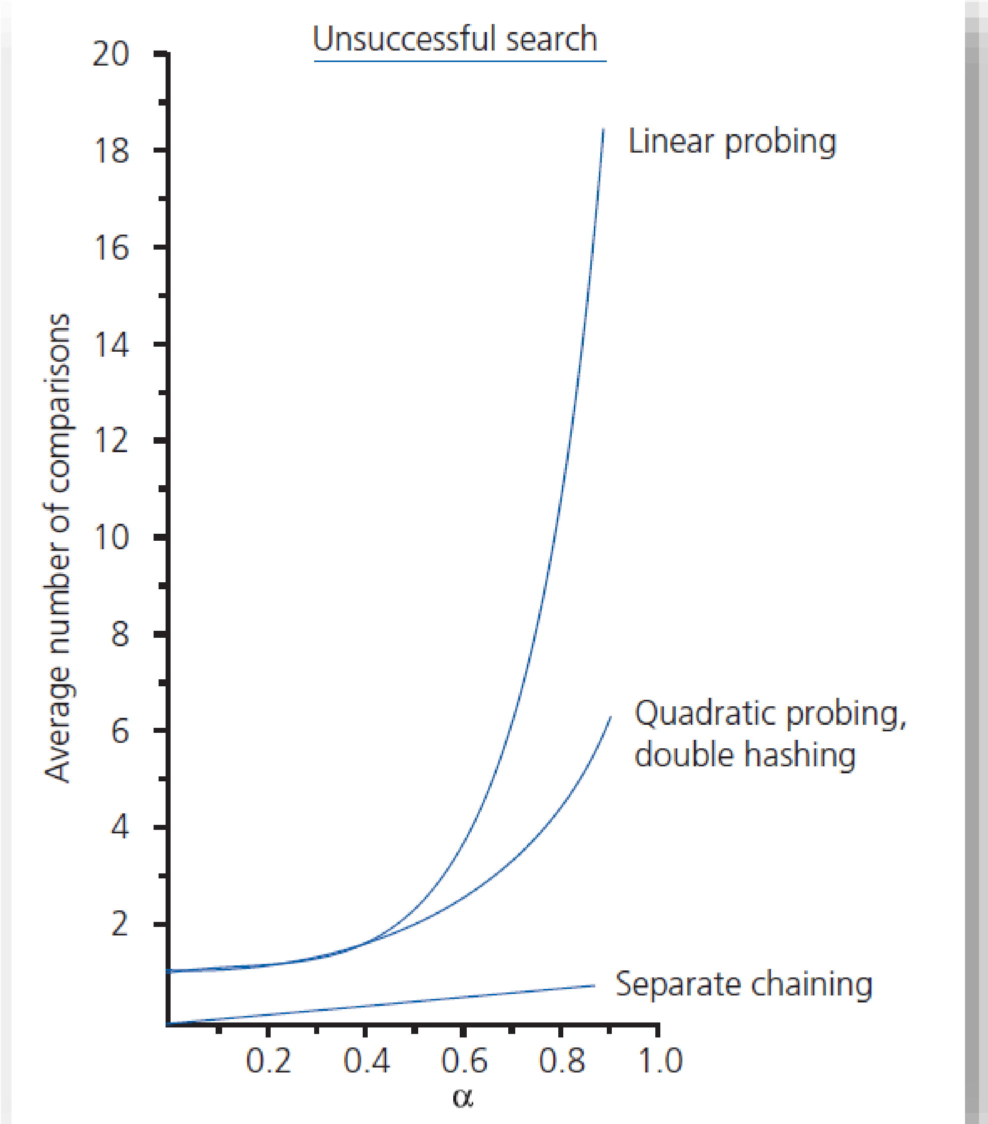
The Efficiency of Hashing

- The relative efficiency of four collision-resolution methods



The Efficiency of Hashing

- The relative efficiency of four collision-resolution methods

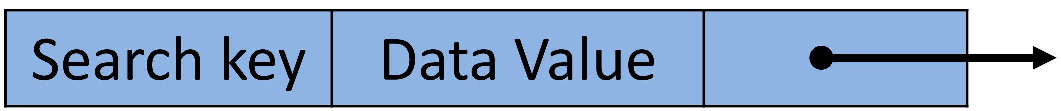


What Constitutes a Good Hash Function?

- Is hash function easy and fast to compute?
- Does hash function scatter data evenly throughout hash table?
- How well does hash function scatter random data?
- How well does hash function scatter *non-*random data?

Dictionary Traversal: An Inefficient Operation Under Hashing

- Entries hashed into table[i] and table[i+1] have no ordering relationship
- Hashing does not support well traversing a dictionary in sorted order
  - Generally better to use a search tree
- In external storage possible to see
  - Hashing implementation of getValue
  - And search-tree for ordered operations simultaneously



- A dictionary entry when separate chaining is used

Using Hashing, Separate Chaining to Implement ADT Dictionary

```
/** A class of entry objects for a hashing implementation of the
    ADT dictionary.
 @file HashedEntry.h */

#ifndef HASHED_ENTRY_
#define HASHED_ENTRY_

#include "Entry.h"

template<class KeyType, class ValueType>
class HashedEntry : public Entry<KeyType, ValueType>
{
private:
   std::shared_ptr<HashedEntry<KeyType, ValueType>> nextPtr;
public:
```

- The class HashedEntry

```
   std::shared_ptr<HashedEntry<KeyType, ValueType>> nextPtr;
public:
   HashedEntry();
   HashedEntry(KeyType searchKey, ValueType newValue);
   HashedEntry(KeyType searchKey, ValueType newValue,
               std::shared_ptr<HashedEntry<KeyType, ValueType>> nextEntryPtr);

   void setNext(std::shared_ptr<HashedEntry<KeyType, ValueType>>
                                                    nextEntryPtr nextEntryPtr);
   auto getNext() const;
}; // end HashedEntry

#include "HashedEntry.cpp"
#endif
```

- The class HashedEntry