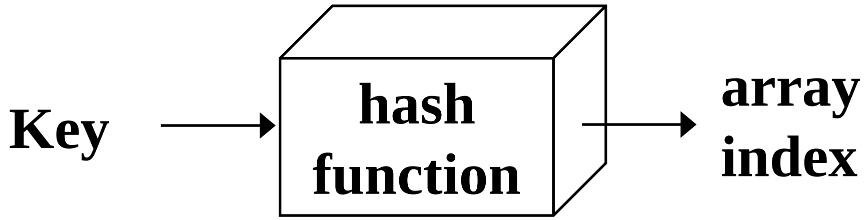
# UCS301: Hashing

# Hash Functions and Hash Tables

- Hashing has 2 major components
  - Hash function ***h***
  - Hash Table Data Structure of size ***N***
- A hash function ***h*** maps keys (a identifying element of record set) to hash value or hash key which refers to specific location in Hash table
- Example:

  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key $x$
- A hash table data structure is an array or array type ADTof some fixed size, containing the keys.
- An array in which records are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*



- ***Hashed key***: the result of applying a hash function to a key
- Keys and entries are scattered throughout the array
- Contains the main advantages of both Arrays and Trees
- Mainly the topic of hashing depends upon the two main factors / parts
  (a) Hash Function (b) Collision Resolution
- Table Size is also an factor (miner) in Hashing, which is 0 to tablesize-1.

# Hash Tables

- **Notation:**
  - $U$ – Universe of all possible keys.
  - $K$ – Set of keys actually stored in the dictionary.
  - $|K| = n$.
- When U is very large,
  - Arrays are not practical.
  - $|K| << |U|$.
- Use a table of size proportional to $|K|$ – The hash tables.
  - However, we lose the direct-addressing ability.
  - Define functions that map keys to slots of the hash table.

# Table Size

- Hash table size
  - Should be appropriate for the hash function used
  - Too big will waste memory; too small will increase collisions and may eventually force *rehashing* (copying into a larger table)

# Hash Function

- The mapping of keys into the table is called ***Hash Function***
- A hash function,
  - Ideally, it should distribute keys and entries evenly throughout the table
  - It should be easy and quick to compute.
  - It should minimize *collisions*, where the position given by the hash function is already occupied
  - It should be applicable to all object
- Different types of hash functions are used for the mapping of keys into tables.

  (a) Division Method

  (b) Mid-square Method

  (c) Folding Method

# 1. Division Method

- Choose a number *m* larger than the number *n* of keys in *k.*
- The number *m* is usually chosen to be a prime no.
- The hash function *H* is defined as,

  H(k) = k(mod m) or H(k) = k(mod m) + 1

- Denotes the remainder, when *k* is divided by *m*
- 2nd formula is used when range is from 1 to m.

- Example:
  Elements are: 3205, 7148, 2345

  Table size: 0 – 99 (prime)
  m = 97 (prime)

  H(3205)= 4, H(7148)=67, H(2345)=17

- **For 2nd formula *add* 1 into the remainders**.

# 2. Folding Method

- The key *k* is partitioned into no. of parts
- Then add these parts together and ignoring the last carry.
- One can also reverse the first part before adding (right or left justified. Mostly right)

$$H(k) = k1 + k2 + \dots\dots\dots + kn$$

- <u>Example:</u>

  H(3205)=32+05=37 ***<u>or</u>*** H(3250)=32+50=82

  H(7148)=71+48=19 ***<u>or</u>*** H(7184)=71+84=55

  H(2345)=23+45=68 ***<u>or</u>*** H(2354)=23+54=77

# 3. Mid-Square Method

- The key k is squared. Then the hash function H is defined as

  $$H(k) = l$$

- The ***l*** is obtained by deleting the digits from both ends of $K^2$.

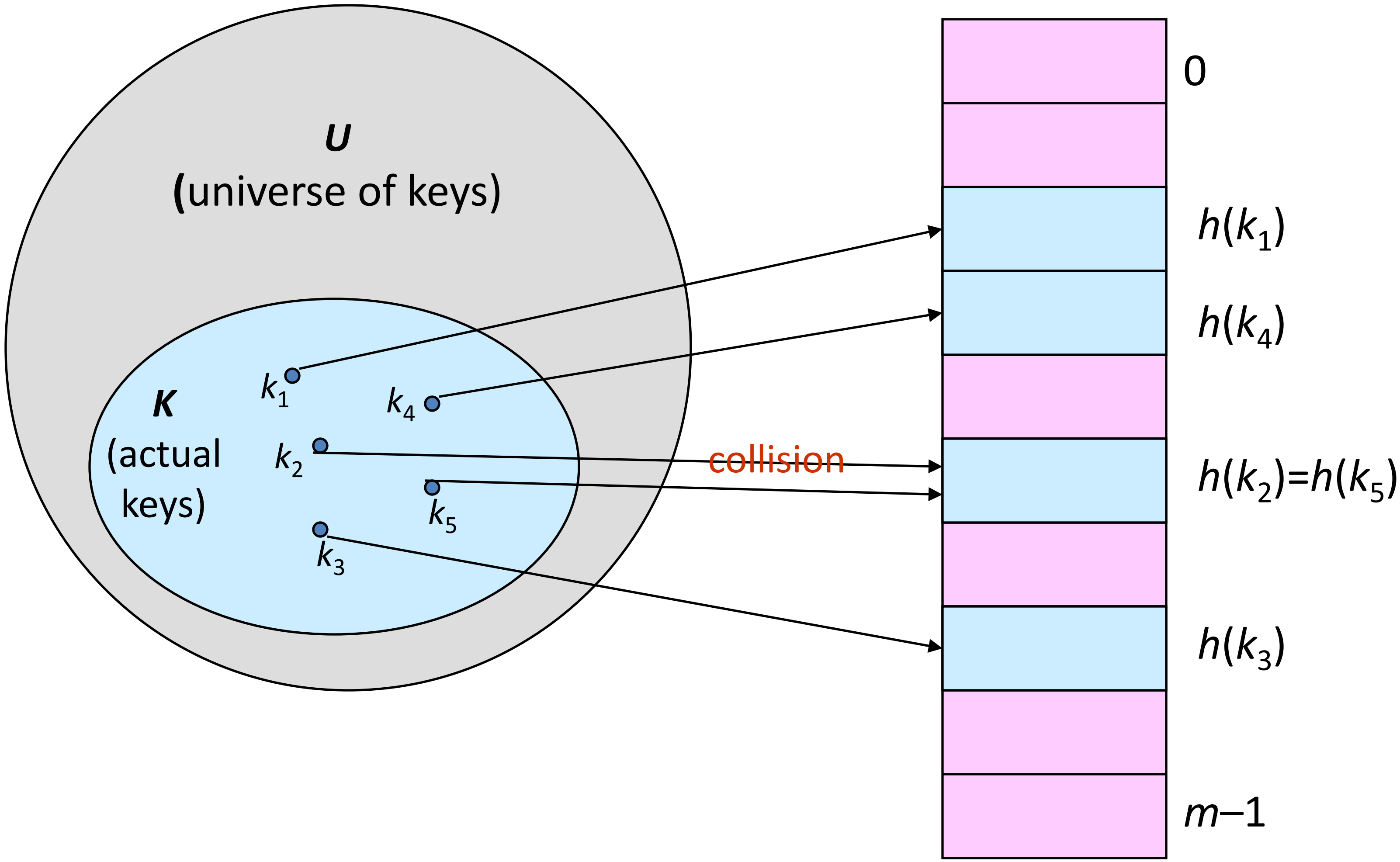
- The same position must be used for all the keys.

- <u>Example:</u>

| | | | |
|---|---|---|---|
| k: | 3205 | 7148 | 2345 |
| $k^2$: | 10272025 | 51093904 | 5499025 |
| H(k): | 72 | 93 | 99 |

- $4^{th}$ and $5^{th}$ digits have been selected. From the right side.

# Hashing



*U*
(universe of keys)
*K*
(actual keys)
$k_1$
$k_4$
$k_2$
$k_5$
$k_3$
collision
0
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
$m$–1

# Collision Resolution

- 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.
- As the number of elements in the table increases, the likelihood of a *collision* increases - so make the table **as large as practical**
- There are several methods for dealing with this:
  - **Separate chaining**
  - **Open addressing**
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
- **Probing**: If the table position given by the hashed key is already occupied, increase the position by some amount, until an empty position is found

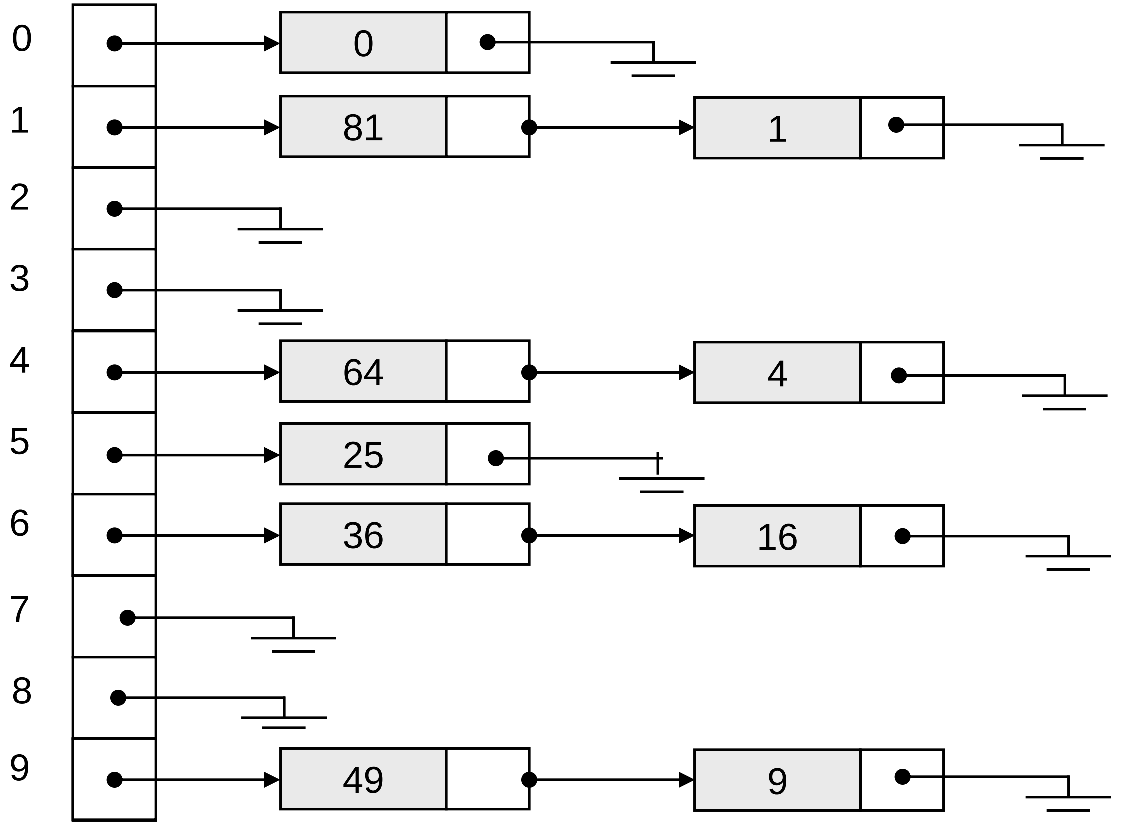
# Separate Chaining

- The idea is to keep a list of all elements that hash to the same value.
  - The array elements are pointers to the first nodes of the lists.
  - A new item is inserted to the front of the list.
- Advantages:
  - Better space utilization for large items.
  - Simple collision handling: searching linked list.
  - Overflow: we can store more items than the hash table size.
  - Deletion is quick and easy: deletion from the linked list.

# Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

hash(key) = key % 10.



# Operations

- **Initialization**: all entries are set to NULL
- **Find**:
  - locate the cell using hash function.
  - sequential search on the linked list in that cell.
- **Insertion**:
  - Locate the cell using hash function.
  - (If the item does not exist) insert it as the first item in the list.
- **Deletion**:
  - Locate the cell using hash function.
  - Delete the item from the linked list.

# Analysis of Separate Chaining

- Collisions are very likely.
  - How likely and what is the average length of lists?
- Load factor $\lambda$ definition:
  - Ratio of number of elements (N) in a hash table to the hash *TableSize*.
    - i.e. $\lambda$ = *N/TableSize*
  - The average length of a list is also $\lambda$
  - For chaining, $\lambda$ is not bound by 1; it can be > 1.

# Cost of searching

- **Cost** = Constant time to evaluate the hash function + time to traverse the list.
- **Unsuccessful search**:
  - We have to traverse the entire list, so we need to compare $\lambda$ nodes on the average.
- **Successful search**:
  - List contains the one node that stores the searched item + 0 or more other nodes.
  - Expected # of other nodes = x = (N-1)/M which is essentially $\lambda$, since M is presumed large.
  - On the average, we need to check *half* of the *other nodes* while searching for a certain element
  - Thus average search cost = 1 + $\lambda$/2

# Summary

- The analysis shows us that the table size is not really important, but the load factor is.
- TableSize should be as *large* as the number of expected elements in the hash table.
  - To keep load factor around 1.
- TableSize should be *prime* for even distribution of keys to hash table cells.

# Linear Probing

- Linear probing uses the hash function

  $h(k,i)=(h'(k)+ i) \bmod m$, for $i=0,1,2,\ldots,m-1$.

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.
  - i.e. *f* is a linear function of *i*, typically *f(i)= I*, where $h(k,i)=(h'(k)+ i) \bmod m$ can be written as $h(k,i)=(h'(k)+ f(i)) \bmod m$.
- $h(k,i)=(h'(k)+ c_1i+c_2i^2) \bmod m$, *where h'* is an auxiliary hash function, $c_1$ and $c_2$ are positive auxiliary constants, and $i=0,1,2,\ldots,m-1$.
- Example:
  - Insert items with keys: 89, 18, 49, 58, 9 into an empty hash table.
  - Table size is 10.
  - Hash function is hash(x) = x mod 10.
    - *f(i) = i*;

**Figure 20.4**
Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

| | *After insert 89* | *After insert 18* | *After insert 49* | *After insert 58* | *After insert 9* |
|---|---|---|---|---|---|
| 0 | | | 49 | 49 | 49 |
| 1 | | | | 58 | 58 |
| 2 | | | | | 9 |
| 3 | | | | | |
| 4 | | | | | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | 18 | 18 | 18 | 18 |
| 9 | 89 | 89 | 89 | 89 | 89 |

# Find and Delete

- The find algorithm follows the same probe sequence as the insert algorithm.
  - A find for 58 would involve 4 probes.
  - A find for 19 would involve 5 probes.
- We must use *lazy deletion* (i.e. marking items as deleted)
  - Standard deletion (i.e. physically removing the item) cannot be performed.
  - e.g. remove 89 from hash table.

# Clustering Problem

- As long as table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Worse, even if the table is relatively empty, blocks of occupied cells start forming.
- This effect is known as *primary clustering*.
- Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

# Quadratic Probing

- Quadratic Probing eliminates primary clustering problem of linear probing.
- Quadratic probing uses a hash function of the form

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

  *where h'* is an auxiliary hash function, $c_1$ and $c_2$ are positive auxiliary constants, and i=0,1,2,….,m-1. Frequently, the following settings are used: $c_1=0$ and $c_2=1$ .
- Collision function is quadratic.
  - The popular choice is $f(i) = i^2$ (Applying $c_1=0$ and $c_2=1$ to $f(i)= c_1 i+c_2 i^2$ )
- If the hash function evaluates to h and a search in cell h is inconclusive, we try cells $h + 1^2$, $h+2^2$, … $h + i^2$. Here, $c_1=0$ and $c_2=1$.
  - i.e. It examines cells 1,4,9 and so on away from the original probe.
- Remember that subsequent probe points are a quadratic number of positions from the *original probe point*.

**Figure 20.6**
A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number). Here, N=10, $c_1$=0 and $c_2$=1.

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

| | *After insert 89* | *After insert 18* | *After insert 49* | *After insert 58* | *After insert 9* |
|---|---|---|---|---|---|
| 0 | | | 49 | 49 | 49 |
| 1 | | | | | |
| 2 | | | | 58 | 58 |
| 3 | | | | | 9 |
| 4 | | | | | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | 18 | 18 | 18 | 18 |
| 9 | 89 | 89 | 89 | 89 | 89 |

# Quadratic Probing

- Problem:
  - We may not be sure that we will probe all locations in the table (i.e. there is no guarantee to find an empty cell if table is more than half full.)
  - If the hash table size is not prime this problem will be much severe.
- However, there is a theorem stating that:
  - If the table size is *prime* and load factor is not larger than 0.5, all probes will be to different locations and an item can always be inserted.

# Theorem

- If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

# Some considerations

- How efficient is calculating the quadratic probes?
  - Linear probing is easily implemented. Quadratic probing appears to require * and % operations.
  - However by the use of the following trick, this is overcome:
    - $H_i = H_{i-1} + 2i - 1 \pmod{M}$

- What happens if load factor gets too high?
  - Dynamically expand the table as soon as the load factor reaches 0.5, which is called *rehashing*.
  - Always double to a prime number.
  - When expanding the hash table, reinsert the new table by using the new hash function.

# Analysis of Quadratic Probing

- Quadratic probing has not yet been mathematically analyzed.
- Although quadratic probing eliminates primary clustering, elements that hash to the same location will probe the same alternative cells. This is know as *secondary clustering*.
- Techniques that eliminate secondary clustering are available.
  - the most popular is *double hashing*.

# Double Hashing

- Double hashing uses a hash function of the form

  $h(k,i)=(h_1(k)+ i\ h_2(k)) \bmod m$, for i=0,1,2,….,m-1.
- A second hash function is used to drive the collision resolution.
  - $f(i) = i * h_2(x)$
- We apply a second hash function to x and probe at a distance $h_2(x)$, $2*h_2(x)$, … and so on.
- The function $h_2(x)$ must never evaluate to zero.
  - e.g. Let $h_2(x) = x \bmod 9$ and try to insert 99 in the previous example.
- A function such as $h_2(x) = R – ( x \bmod R)$ with R a prime smaller than Table Size will work well.
  - e.g. try R = 7 for the previous example.(7 - x mode 7)
- The table size ***N*** must be a prime to allow probing of all the cells
- Common choice of compression map for the secondary hash function: $h_2(x) = x \bmod m$. But, some times $m$ is replaced with other prime number $q$ where q < N.

# Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h_1(k) = k \bmod 13$
  - $h_2(k) = k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

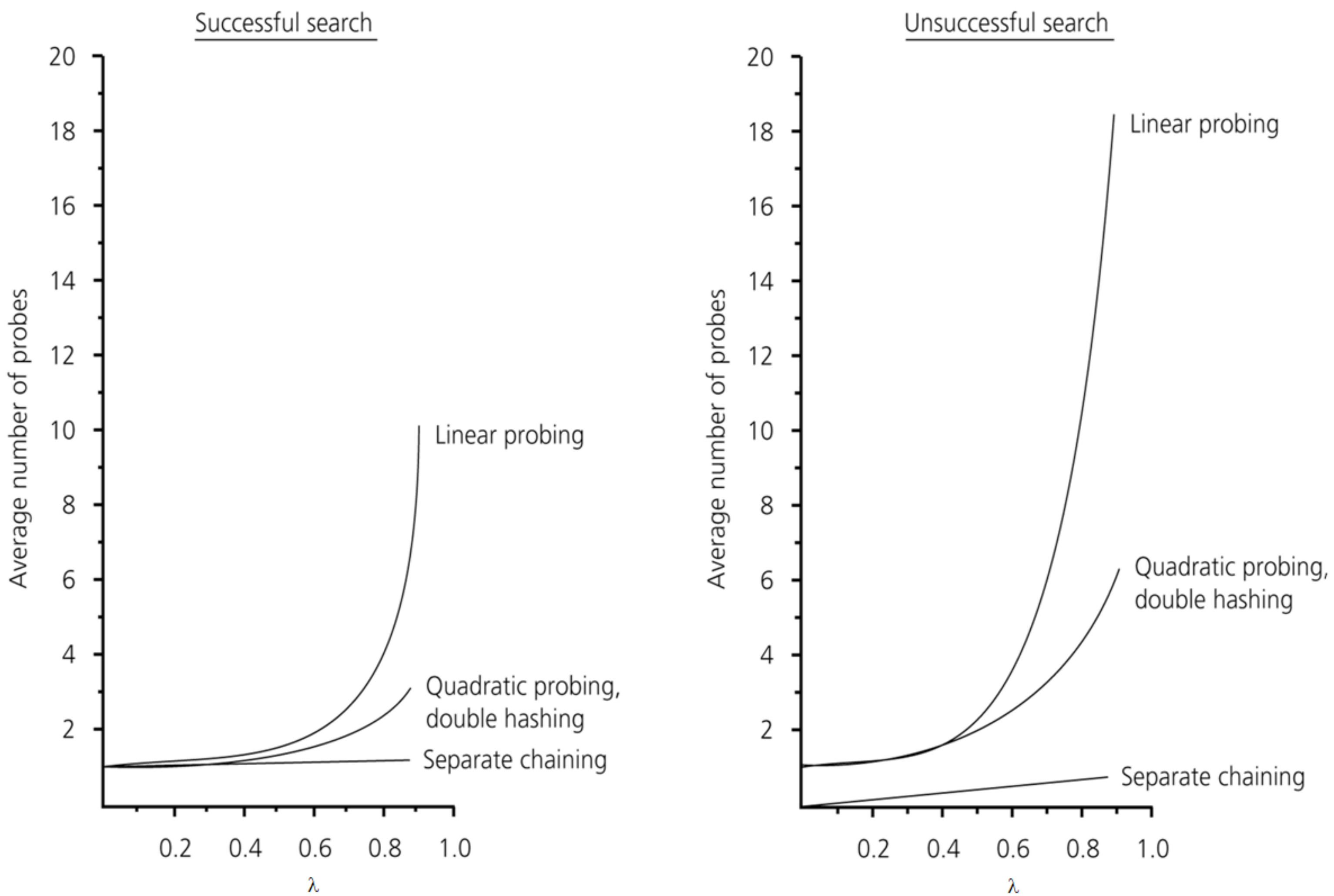
| $k$ | $h_1(k)$ | $h_2(k)$ | Probes | | |
|---|---|---|---|---|---|
| 18 | 5 | 9 | 5 | | |
| 41 | 2 | 8 | 2 | | |
| 22 | 9 | 10 | 9 | | |
| 44 | 5 | 7 | 5 | 7 | |
| 59 | 7 | 10 | 7 | 10 | 0 |
| 32 | 6 | 4 | 6 | | |
| 31 | 5 | 8 | 5 | 8 | |
| 73 | 8 | 11 | 8 | 11 | |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

$\Downarrow$

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 59 | | 41 | | | 18 | 32 | 44 | 8 | 22 | 44 | 11 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# The relative efficiency of four collision-resolution methods



# Applications of Hashing

- Compilers use hash tables to keep track of declared variables

- A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time

- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again

- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different