CS-2001
DATA STRUCTURE

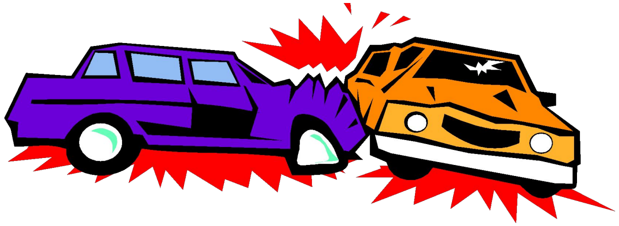
Dr. Hashim Yaseen

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.

HASHING

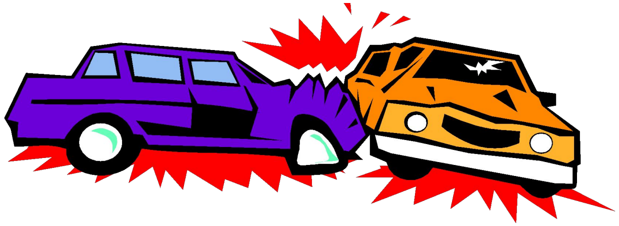
Collision



- The condition resulting when two or more keys produce the same hash location.

- *A good hash function minimizes collisions* by spreading the elements uniformly throughout the array.

Collision



- Collision handling techniques
  - Linear Probing
  - Rehashing
  - Double Hashing
  - Quadratic Probing
  - Random Probing
  - Buckets
  - Chaining

Collision Resolution Techniques

- There are two broad ways of collision resolution:

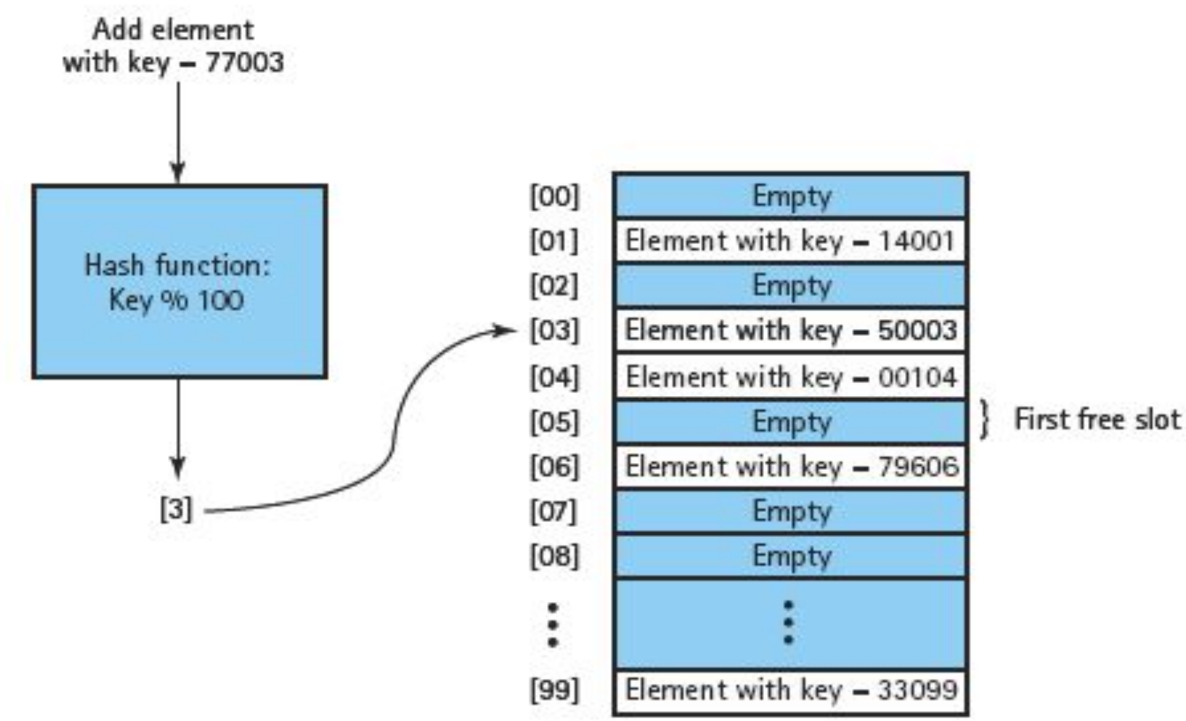
1. **Open Addressing:** Array-based implementation.

   (i) Linear probing (linear search)
   (ii) Quadratic probing (nonlinear search)
   (iii) Double hashing (uses two hash functions)

2. **Separate Chaining:** A linked list implementation

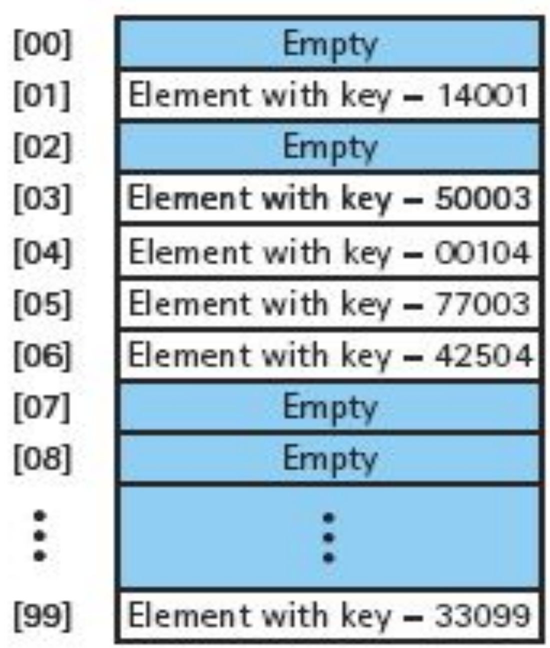
Linear Probing

- *Resolving a hash collision by sequentially searching* a hash table beginning at the location return by the hash function.

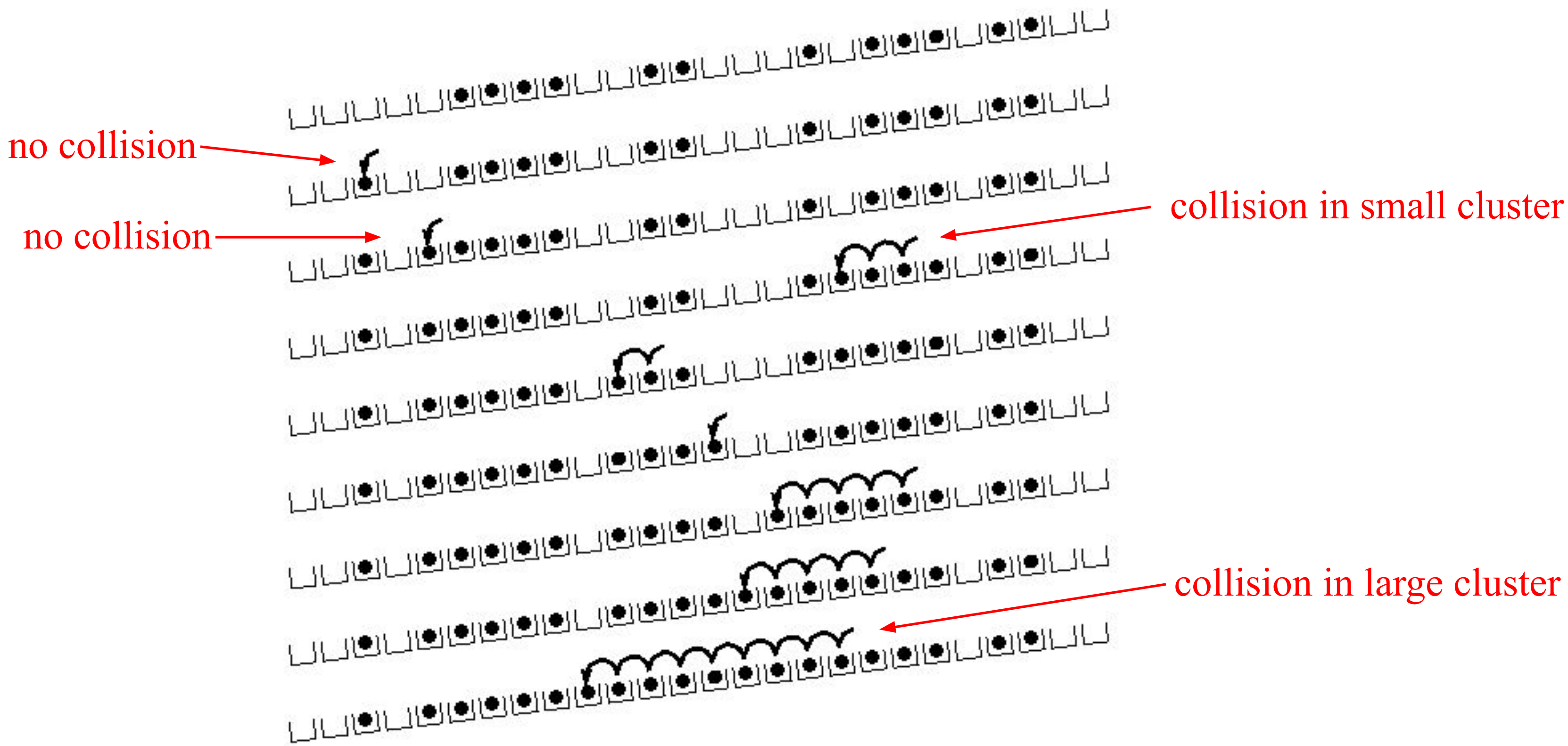


Linear Probing – Clustering

- *The tendency of elements to become unevenly distributed in the hash table, with many elements clustering around a single hash location.*



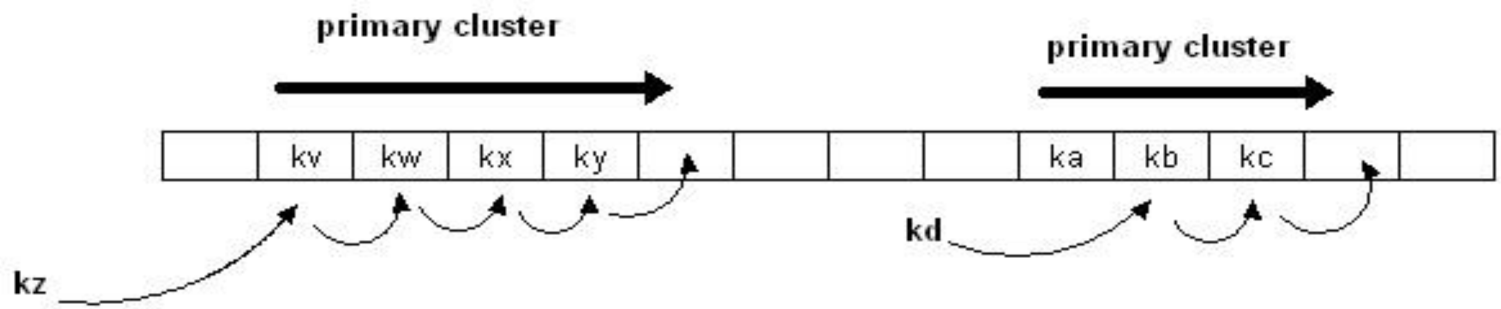
Linear Probing – Clustering



Disadvantage of Linear Probing: Primary Clustering

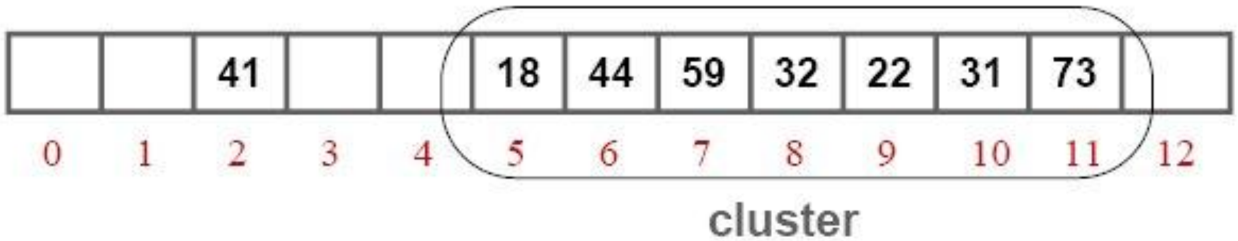
- Linear probing is subject to a **primary clustering** phenomenon.
- Elements tend to cluster around table locations that they originally hash to.
- *Primary clusters can combine to form larger clusters.*
  - This leads to long probe sequences and hence deterioration in hash table efficiency.

Disadvantage of Linear Probing: Primary Clustering



Example of a primary cluster: Insert keys: **18, 41, 22, 44, 59, 32, 31, 73**, in this order, in an originally empty hash table of size **13**, using the hash function **h(key) = key % 13** and **c(i) = i**:

h(18) = 5
h(41) = 2
h(22) = 9
h(44) = 5+1
h(59) = 7
h(32) = 6+1+1
h(31) = 5+1+1+1+1+1
h(73) = 8+1+1+1



REHASHING

Rehashing

- Hash Table may get full
  - No more insertions possible
- Hash table may get *too* full
  - Insertions, deletions, search take longer time
- Solution: Rehash
  - Build another table that is twice as big and has a new hash function
  - Move all elements from smaller table to bigger table
- Cost of Rehashing = ***O(N)***
  - But happens only when table is close to full
  - Close to full = table is X percent full, where X is a tunable parameter

Rehashing Example

Original Hash Table

0	6
1	15
2	
3	24
4	
5	
6	13

After Inserting 23

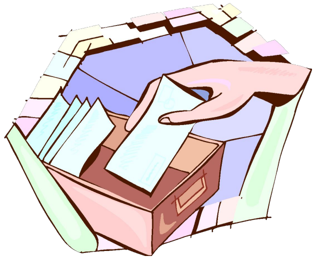
0	6
1	15
2	23
3	24
4	
5	
6	13

After Rehashing

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

DOUBLE HASHING

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: avoids clustering
- Disadvantage:
  - harder to delete an element
  - Can generate m^2 probe sequences maximum

Double Hashing: Example

$h_1(k) = k \bmod \mathbf{13}$

$h_2(k) = 1 + (k \bmod \mathbf{11})$

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

- **Insert key 14:**

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

$h_2(14,0) = 1 + (14 \bmod 11) = 4$

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

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

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

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

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

Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

A good choice for g is to choose a prime R < TableSize

and let $g(k) = R - (k \bmod R)$.

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1*g(k)) mod TableSize

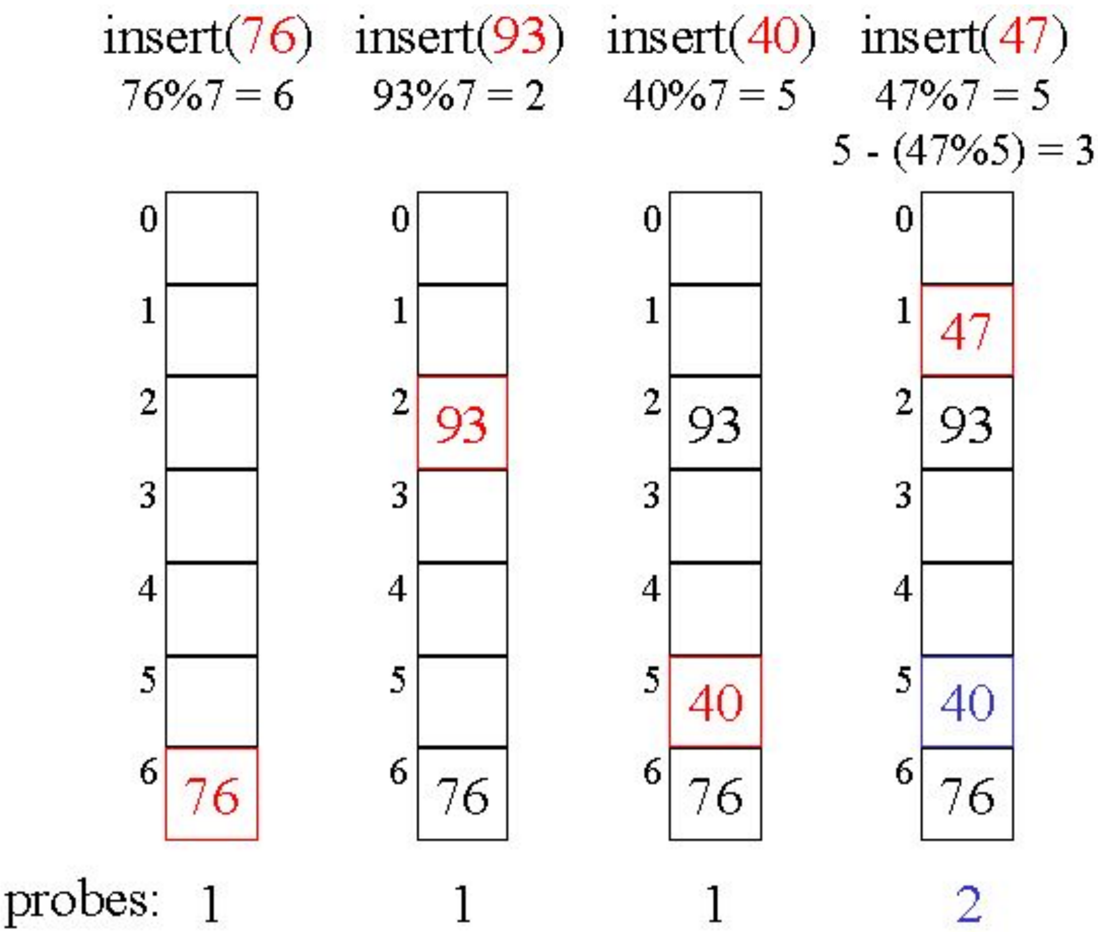
  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

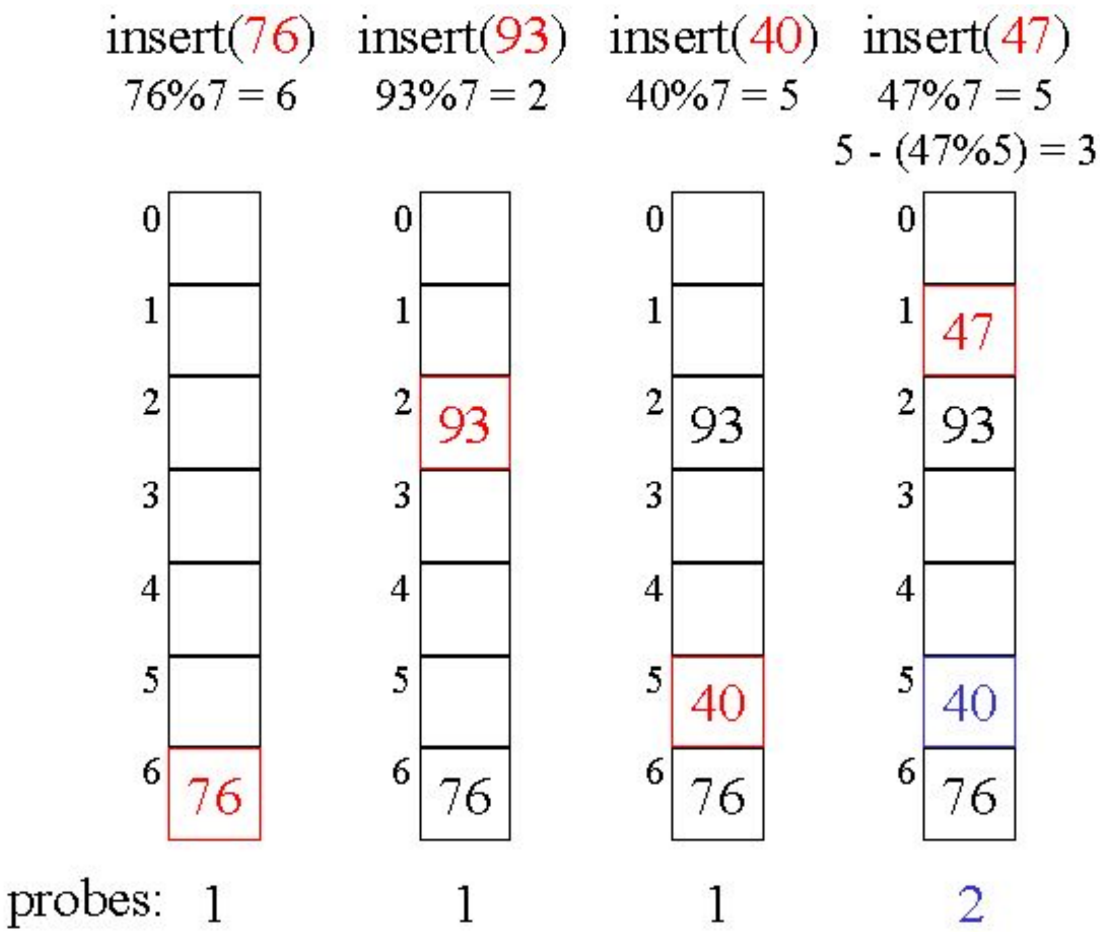
  . . .

  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double Hashing … Example 1



Double Hashing … Example 1

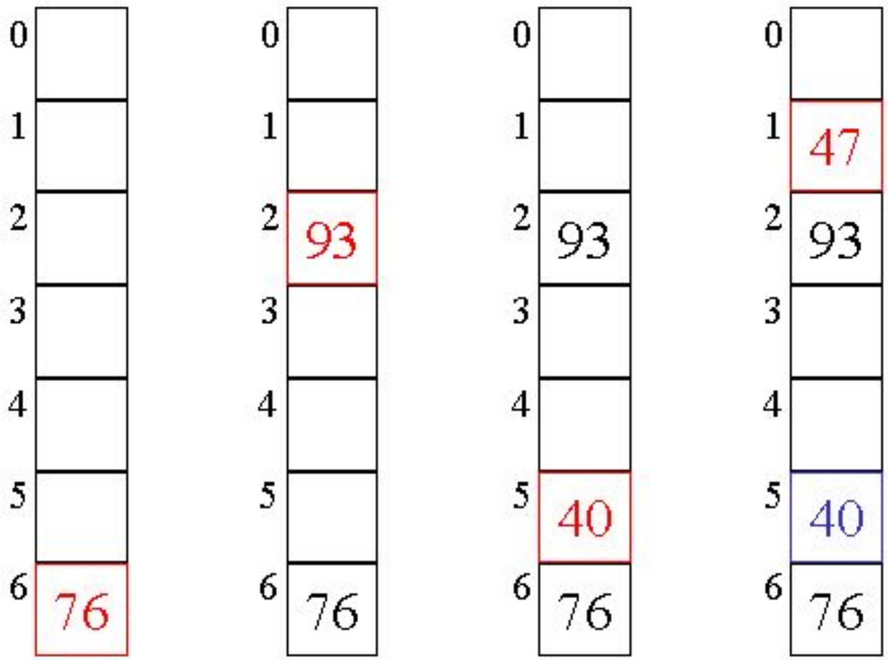


Double Hashing … Example 1

insert(76) insert(93) insert(40) insert(47) insert(10) insert(55)

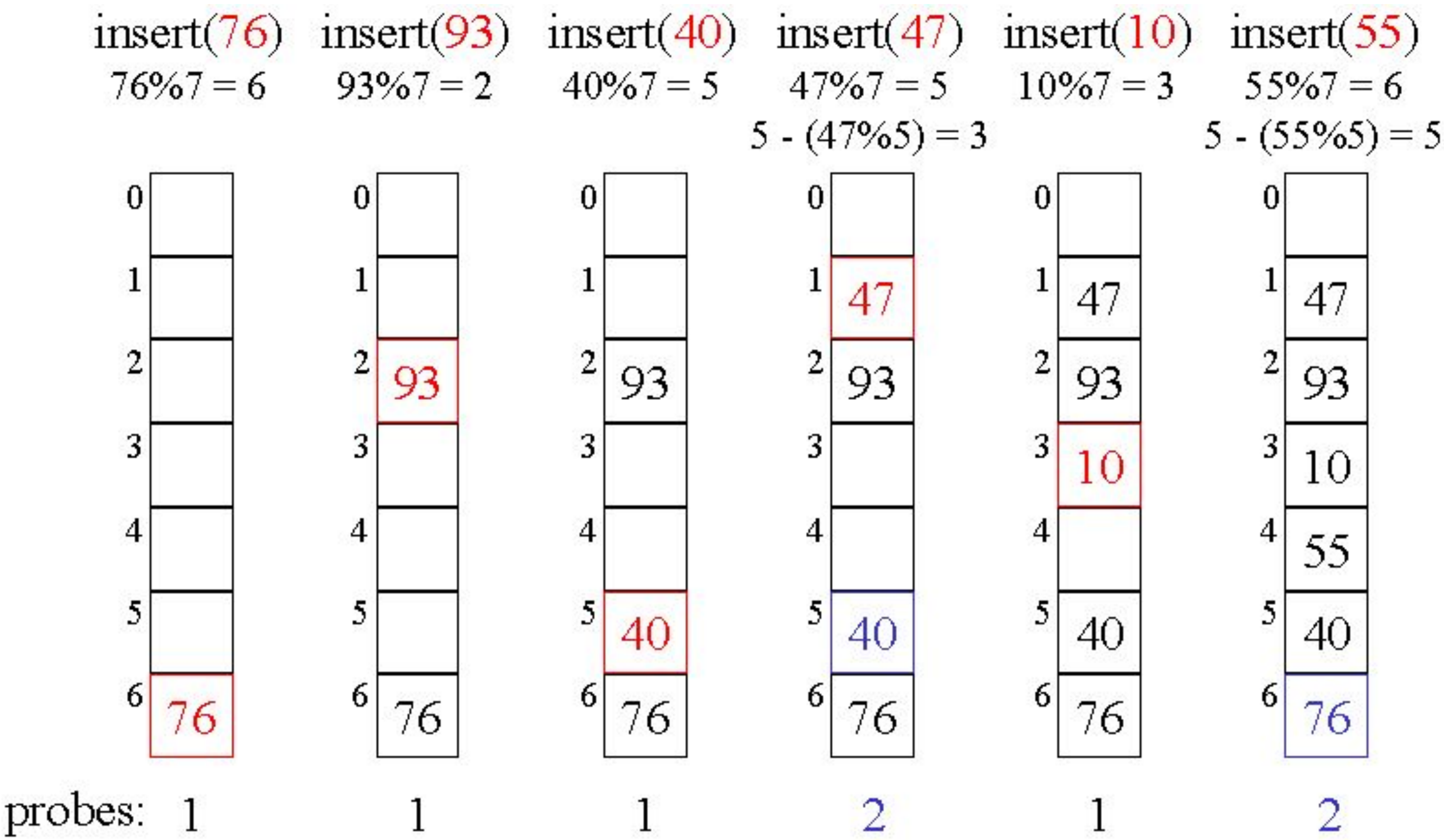
76%7 = 6 93%7 = 2 40%7 = 5 47%7 = 5

5 - (47%5) = 3



probes: 1 1 1 2

Double Hashing … Example 1



Double Hashing … Example 2

Example: Load the keys **18, 26, 35, 9, 64, 47, 96, 36, and 70** in this order, in an empty hash table of size **13**

(a) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: $\mathbf{h_p(key) = 1 + key \% 12}$

(b) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: $\mathbf{h_p(key) = 7 - key \% 7}$

Show all computations.

Example-2

$h_i(key) = [h(key) + i*h_p(key)]\% 13$

$h(key) = key \% 13$

$h_p(key) = 1 + key \% 12$

$h_0(18) = 18\%13 = 5$

$h_0(26) = 26\%13 = 0$

$h_0(35) = 35\%13 = 9$

$h_0(9) = 9\%13 = 9$ collision

$h_p(9) = (1 + 9)\%12 = 10$

$h_1(9) = (9 + 1*10)\%13 = 6$

$h_0(64) = 64\%13 = 12$

$h_0(47) = 47\%13 = 8$

0	1	2	3	4	5	6	7	8	9	10	11	12
26			70		18	9	96	47	35	36		64

Example-2

$h_i(key) = [h(key) + i*h_p(key)]\% 13$

$h(key) = key \% 13$

$h_p(key) = 1 + key \% 12$

$h_0(96) = 96\%13 = 5$ collision

$h_p(96) = (1 + 96)\%12 = 1$

$h_1(96) = (5 + 1*1)\%13 = 6$ collision

$h_2(96) = (5 + 2*1)\%13 = 7$

$h_0(36) = 36\%13 = 10$

$h_0(70) = 70\%13 = 5$ collision

$h_p(70) = 1 + 70\%12 = 11$

$h_1(70) = (5 + 1*11)\%13 = 3$

0	1	2	3	4	5	6	7	8	9	10	11	12
26			70		18	9	96	47	35	36		64

Double Hashing

Performance of Double hashing:

- Much better than linear or quadratic probing because it *eliminates both primary and secondary clustering*.
- BUT it requires the computation of a second hash function $\mathbf{h_p}$.

QUADRATING PROBING

Quadrating Probing

- Resolving a hash collision by using rehashing formula,

  $(\text{HashValue} \pm I^2)\%\text{array\_size}$,

  - Where I is the number of times that the rehash function has been applied
- It distributes the key on a wide range over the hash table.
- Quadratic probing reduces clustering.

Quadrating Probing

$f(i) = i^2$

Less likely to encounter Primary Clustering

- Probe sequence:
  - 0^{th} probe = h(k) mod TableSize
  - 1^{th} probe = (h(k) + 1) mod TableSize
  - 2^{th} probe = (h(k) + 4) mod TableSize
  - 3^{th} probe = (h(k) + 9) mod TableSize
  - . . .
  - i^{th} probe = $(h(k) + i^2)$ mod TableSize

Quadrating Probing

Example:

- Load the keys **23, 13, 21, 14, 7, 8, and 15**, in this order, in a hash table of size **7** using quadratic probing with $\mathbf{c(i) = \pm i^2}$ and the hash function: **h(key) = key % 7**
- The required probe sequences are given by:

$$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$$

Quadrating Probing

keys **23, 13, 21, 14, 7, 8, and 15,**

$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$

$h_0(23) = (23 \pm 0) \% 7 = 2$

$h_0(13) = (13 \pm 0) \% 7 = 6$

$h_0(21) = (21 \pm 0) \% 7 = 0$

$h_0(14) = (14 \pm 0) \% 7 = 0$ **collision**

$h_1(14) = (14 + 1^2) \% 7 = 1$

$h_0(7) = (7 \pm 0) \% 7 = 0$ **collision**

$h_1(7) = (7 + 1^2) \% 7 = 1$ **collision**

$h_{-1}(7) = (7 - 1^2) \% 7 = 6$ **collision**

$h_2(7) = (7 + 2^2) \% 7 = 4$

0	21
1	14
2	23
3	
4	7
5	
6	13

Quadrating Probing

keys **23, 13, 21, 14, 7, 8, and 15,**

$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$

$h_0(8) = (8 \pm 0)\%7 = 1$ collision

$h_1(8) = (8 + 1^2) \% 7 = 2$ collision

$h_{-1}(8) = (8 - 1^2) \% 7 = 0$ collision

$h_2(8) = (8 + 2^2) \% 7 = 5$

$h_0(15) = (15 \pm 0)\%7 = 1$ collision

$h_1(15) = (1\ 5 + 1^2) \% 7 = 2$ collision

$h_{-1}(15) = (15 - 1^2) \% 7 = 0$ collision

$h_2(15) = (15 + 2^2) \% 7 = 5$ collision

$h_{-2}(15) = (15 - 2^2) \% 7 = 4$ collision

$h_3(15) = (15 + 3^2)\%7 = 3$

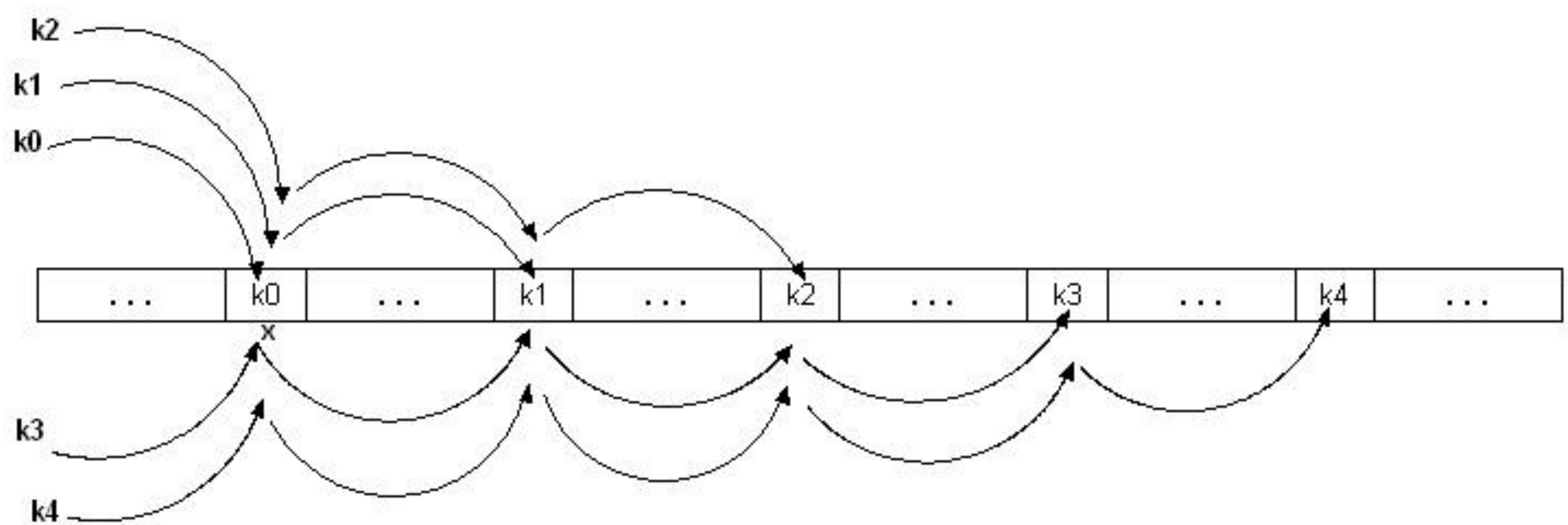
0	21
1	14
2	23
3	15
4	7
5	8
6	13

Quadrating Probing

- Quadratic probing is **better than linear probing** because it eliminates primary clustering.
- However, it may result in **secondary clustering**: if **h(k1) = h(k2)** the probing sequences for **k1** and **k2** are exactly the same. This sequence of locations is called a secondary cluster.
- Secondary clustering is less harmful than primary clustering because secondary clusters do not combine to form large clusters.
- **Example of Secondary Clustering:** Suppose keys **k0, k1, k2, k3, and k4** are inserted in the given order in an originally empty hash table using **quadratic probing** with **c(i) = i^2**.
- Assuming that each of the keys hashes to the same array index **x**. A secondary cluster will develop and grow in size:

Quadrating Probing

- **Example of Secondary Clustering:** Suppose keys **k0, k1, k2, k3, and k4** are inserted in the given order in an originally empty hash table using **quadratic probing** with **c(i) = i^2**.
- Assuming that each of the keys hashes to the same array index **x**. A secondary cluster will develop and grow in size:

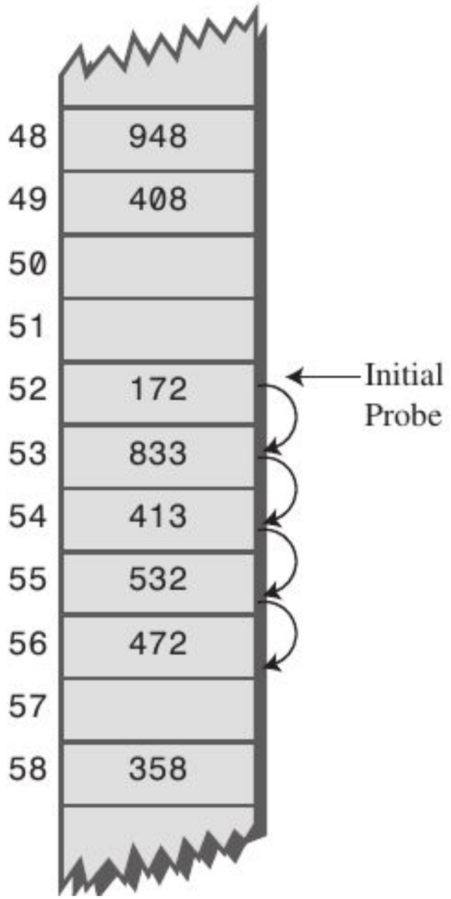


Quadrating Probing

Primary Clustering vs Secondary Clustering

Primary clustering is the tendency for a collision resolution scheme such as linear probing to create long runs of filled slots near the hash position of keys.

Example: If the primary hash index is x, subsequent probes go to x+1, x+2, x+3, and so on, this results in Primary Clustering.

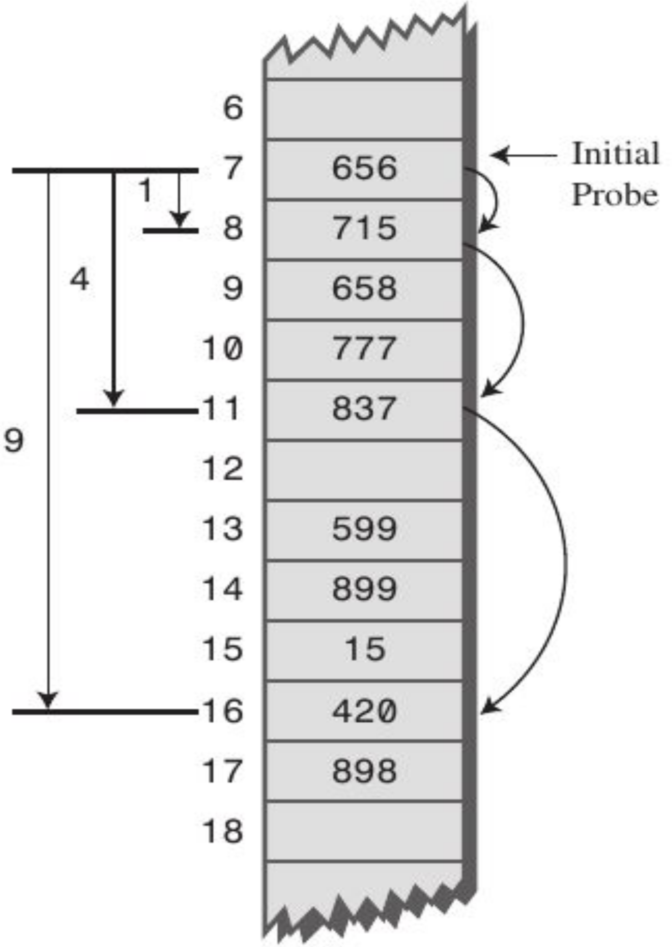


Quadrating Probing

Primary Clustering vs Secondary Cl

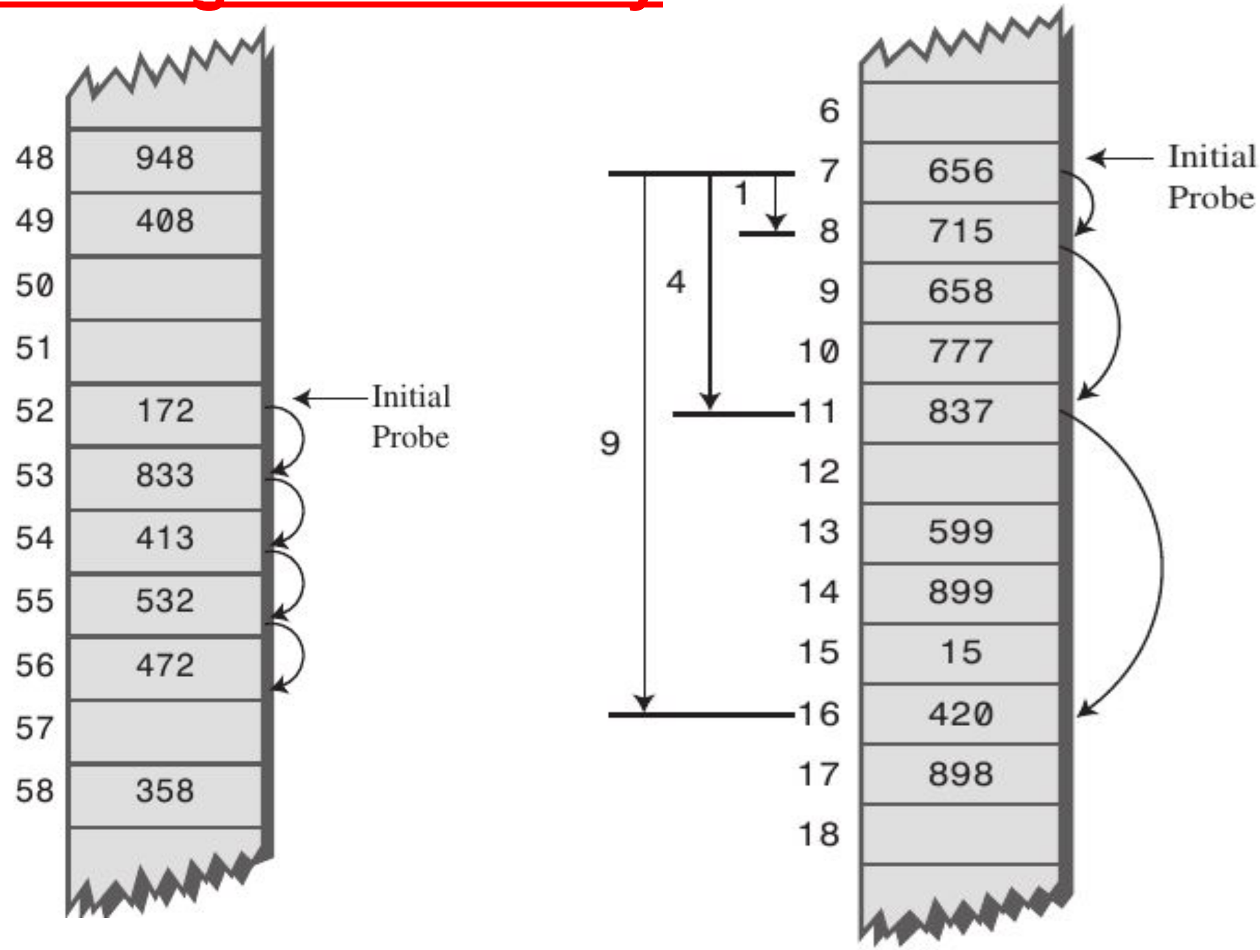
A secondary clustering is a tendency for a collision resolution scheme such as quadratic probing to create long runs of filled slots away from the hash position of keys.

Example: If the primary hash index is x, probes go to x+1, x+4, x+9, x+16, x+25, and so on, this results in Secondary Clustering.



Quadrating Probing

Primary Clustering vs Secondary Clustering



RANDOM PROBING

Random Probing

- Resolving a hash collision by *generating pseudo-random hash values in successive applications of the rehash function*

- Random probing is an excellent technique for eliminating clustering, but it tends to be slower than other techniques.

Random Probing

- Random probing
  - Randomize(X)
  - h0(X) = Hash(X),
  - h1(X) = (h0(X) + RandomGen()) mod TableSize,
  - h2(X) = (h1(X) + RandomGen()) mod TableSize, …
- Use Randomize(X) to ‘seed’ the random number generator using X
- Each call of RandomGen() will return the next random number in the random sequence for seed X

References

- Nell Dale – Chapter 10.
- http://www.cplusplus.com/doc/tutorial/templates/
- Robert Lafore, Chapter 14, Page 681