Hash Tables

Need for Hashing

- Many applications deal with lots of data
  - Web searches, Spell checkers, Databases, Compilers, passwords
  - Bank Applications, Customer search, phone number search etc
- The look ups are time critical
- Search
  - Array O(n)
  - Binary Search O(log n)

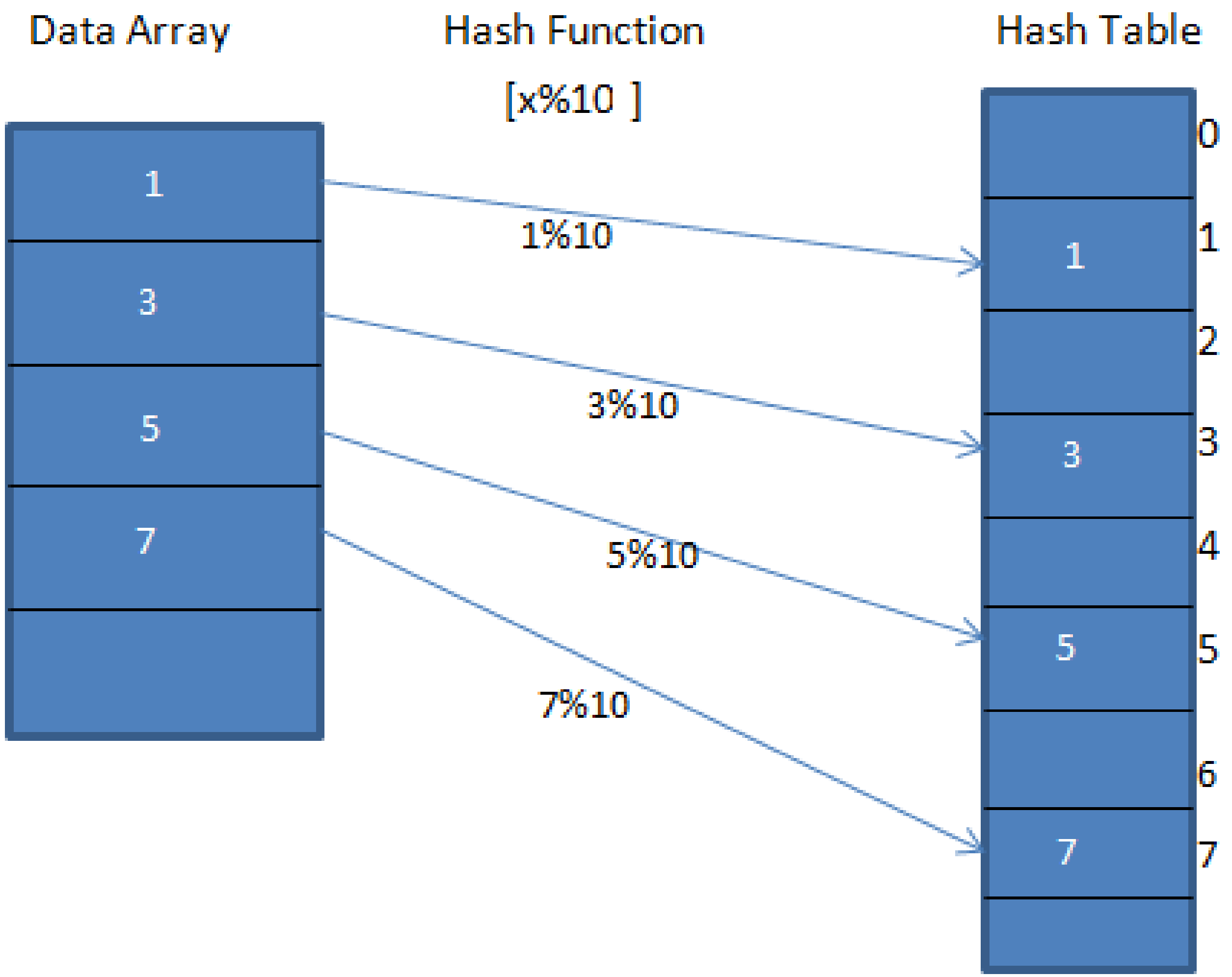
- Better data structure -> O(1)

Searching in O(1) time

Hashing

- Mapping key to address

h(key) = address



Finding a Hash Function

- Assume that N = 5 and the values we need to insert are: cab, bea, bad etc.
- Let a=0, b=1, c=2, etc
- Define H such that
  - H[data] = (Σ characters) Mod N
- H[cab] = (2+0+1) Mod 5 = 3
- H[bea] = (1+4+0) Mod 5 = 0
- H[bad] = (1+0+3) Mod 5 = 4

Choosing a Hash Functions

- A good hash function must
  - Be easy to compute
  - Avoid collisions
- How do we find a good hash function?
- A bad hash function
  - Let S be a string and $H(S) = \Sigma S_i$ where S_i is the i^{th} character of S
  - Why is this bad?

Division Method

The hash function divides the value k by M and then uses the

Formula:

h(K) = k mod M

Here,

k is the key value, and

M is the size of the hash table.

Division Method : Example

- $k = 12345$
  $M = 95$
  $h(12345) = 12345 \bmod 95$
  $= 90$
- $k = 1276$
  $M = 11$
  $h(1276) = 1276 \bmod 11$
  $= 0$

Mid Square Method

It involves two steps to compute the hash value-

1.Square the value of the key k i.e. k^2

2.^{2}Extract the middle **r** digits as the hash value.

Formula:

h(K) = h(k x k)

Here,

k is the key value.

The value of **r** can be decided based on the size of the table.

Mid Square Method - Example

Suppose the hash table has 100 memory locations.

r = 2 because two digits are required to map the key to the memory location.

k = 60

k x k = 60 x 60

 = 3600

h(60) = 60

The hash value obtained is 60

Digit Folding Method

This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
2. Add the individual parts.
3. The hash value is obtained by ignoring the last carry if any.

Formula:

k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ kn

h(K)= s

Here,

s is obtained by adding the parts of the key **k**

Digit Folding Method - Example:

$k = 12345$

$k1 = 12, k2 = 34, k3 = 5$

$s = k1 + k2 + k3$

$\quad = 12 + 34 + 5$

$\quad = 51$

$h(K) = 51$

Multiplication Method

This method involves the following steps:

1. Choose a constant value A such that $0 < A < 1$.
2. Multiply the key value with A.
3. Extract the fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M.
5. The resulting hash value is obtained by taking the floor of the result obtained in step 4.

Formula:

h(K) = floor (M (kA mod 1))

Here,

M is the size of the hash table.

k is the key value.

A is a constant value.

Multiplication Method : Example

k = 12345
A = 0.357840
M = 100

*h(12345) = floor[100 (12345*0.357840 mod 1)]*
= floor[100 (4417.5348 mod 1)]
= floor[100 (0.5348)]
= floor[53.48]
= 53

Properties of Good Hash Functions

- Must return number 0, …, tablesize
- Should be efficiently computable – O(1) time
- Should not waste space unnecessarily
  - For every index, there is at least one key that hashes to it
  - Load factor lambda λ = (number of keys / TableSize)
- Should minimize collisions

  = different keys hashing to same index

Collisions

- Try inserting “abc”, “cba”, “bca”
- H[abc] = 3 [a = 0, b = 1, c = 2]
- H[cba] = 3
- H[bca] = 3
- They all map to the same location
- H is not a good hash map
- This is called “Collision”
- When collisions occur, we need to “handle” them
- Collisions can be reduced with a selection of a good hash function

Collision Resolution

- Separate Chaining
- Linear Probing
- Double Hashing
- Quadratic Probing

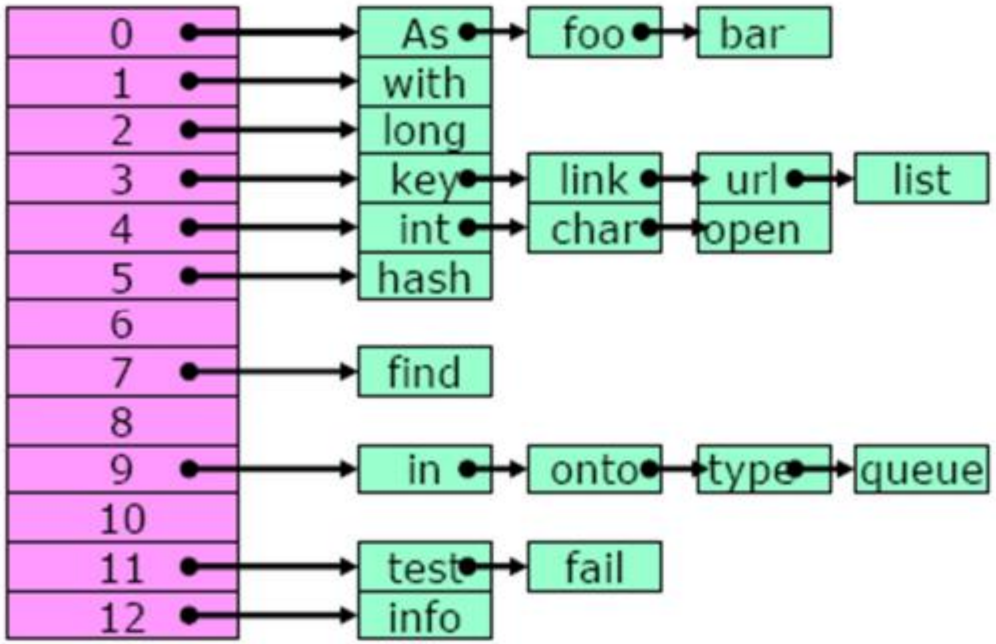
- Separate chaining = Open hashing
- Closed hashing = Open addressing

Open hashing - collisions are stored outside the table

Closed hashing / Open addressing - collisions result in storing one of the records at another slot in the table

Separate Chaining

- Collisions can be resolved by creating a list of keys that map to the same value



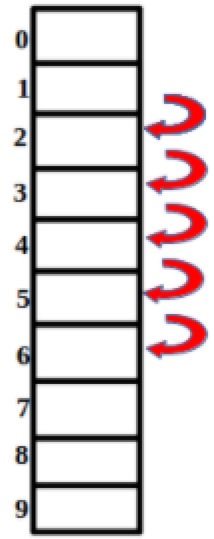
Use an array of N linked lists

where N is the table size

Linear Probing

- The idea: Table remains a simple array of size N
- On insert(x), compute f(x) mod N
- If the cell is full, find another by sequentially searching for the next available slot [f(x)+1, f(x)+2 etc..]
- Linear probing function can be given by

  $$h(x, i) = (f(x) + i) \bmod N \ (i=1,2,\ldots.)$$



Linear Hashing

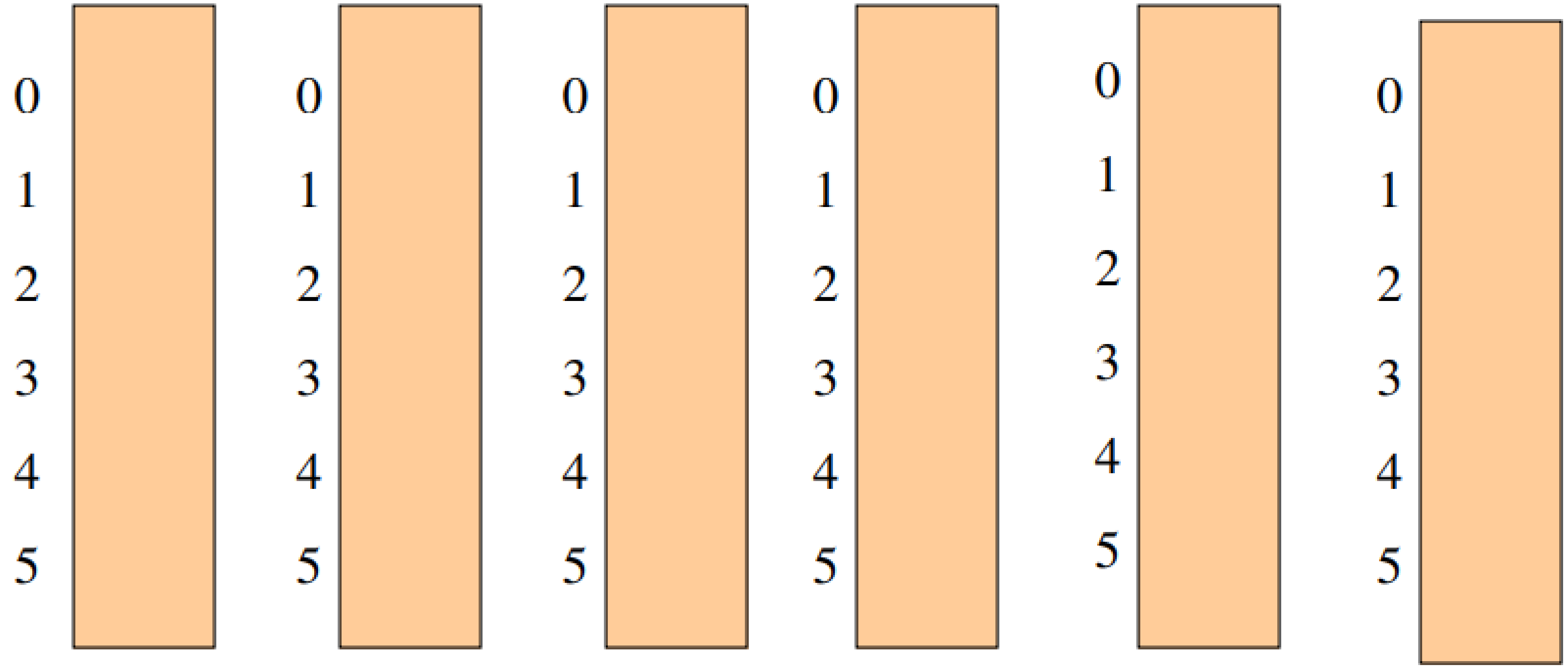
```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

Hash Function
H(k) = k mod 10

	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

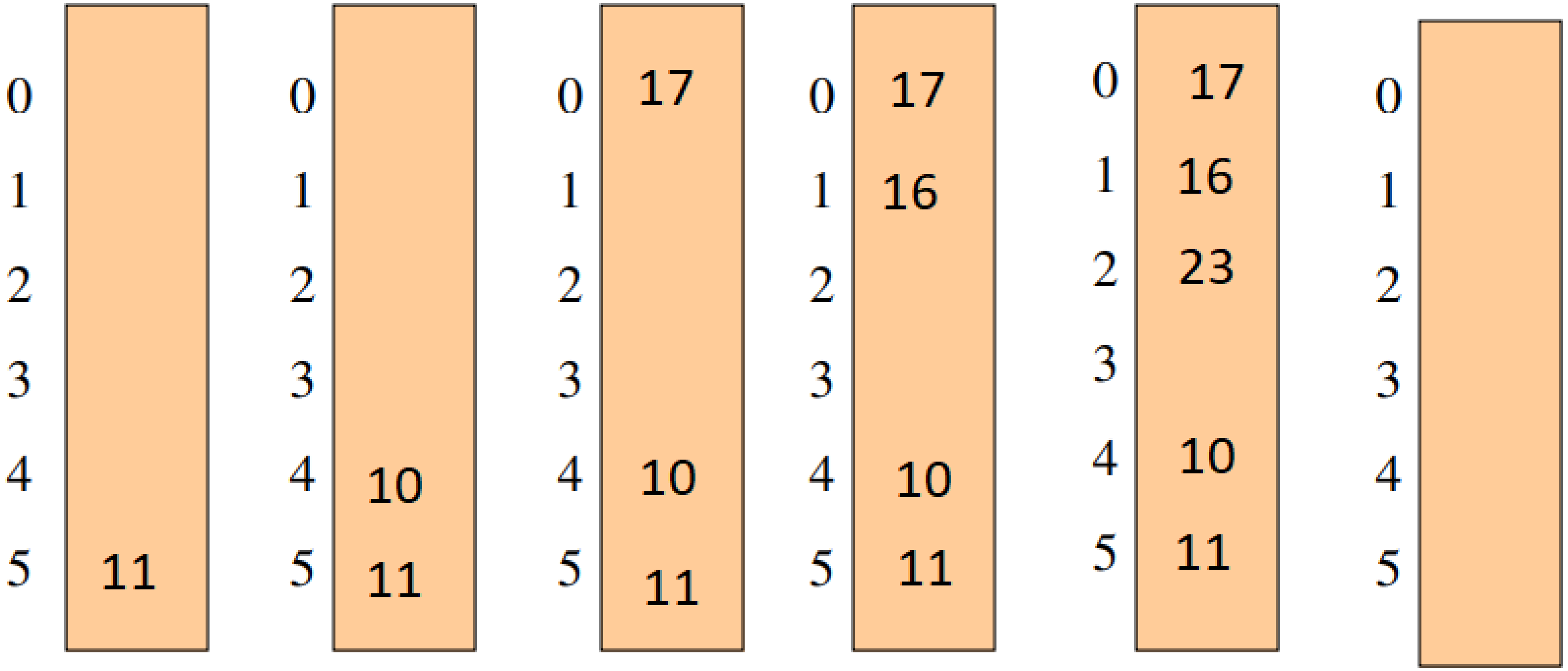
Linear Probing Example

- Consider H(key) = key Mod 6 (assume N=6)
- H(11)=5, H(10)=4, H(17)=5, H(16)=4,H(23)=5
- Draw the Hash table



Linear Probing Example

- Consider H(key) = key Mod 6 (assume N=6)
- H(11)=5, H(10)=4, H(17)=5, H(16)=4,H(23)=5
- Draw the Hash table



Linear Probing - Deletion

- Item in a hash table connects to others in the table
- Deleting items will affect finding the others
- "Lazy Delete" – Just mark the items as inactive rather than removing it.

Linear Probing – Naive Delete

- H(key) = key Mod 6 (assume N=6)
- H(11)=5, H(10)=4, H(17)=5, H(16)=4,H(23)=5
- Delete 17

0	17
1	16
2	23
3	
4	10
5	11

0	
1	16
2	23
3	
4	10
5	11

Deleting 17 leaves gap

Find 23 ?

Linear Probing – Better Deletion

- H(key) = key Mod 6 (assume N=6)
- H(11)=5, H(10)=4, H(17)=5, H(16)=4,H(23)=5
- Delete 17

0	17
1	16
2	23
3	
4	10
5	11

0	XX
1	16
2	23
3	
4	10
5	11

Mark Deleted

Find 23

Load Factor (Open Addressing)

- The load factor λ of a probing hash table is the fraction of the table that is full.
- The load factor ranges from 0 (empty) to 1 (completely full).
- Better to keep the load factor under 0.7
- Double the table size and rehash if load factor gets high
- Cost of Hash function f(x) must be minimized
- When collisions occur, linear probing can always find an empty cell

  But clustering can be a problem

Drawbacks of Linear Probing

- Works until array is full, but as number of items N approaches *TableSize* ($\lambda \approx 1$), access time approaches O(N)
- Very prone to cluster formation. If a key hashes *anywhere* into a cluster, finding a free cell involves going through the entire cluster – and making it grow!
  - *Primary clustering – clusters grow when keys hash to values close to each other*
- Can have cases where table is empty except for a few clusters
  - Does not satisfy good hash function criterion of *distributing keys uniformly*

Quadratic Probing

- A form of Closed Hashing - open addressing
- Main Idea: Spread out the search for an empty slot –

  Increment by i^2 instead of i

- Resolve collisions by examining certain cells (1, 4, 9, …) away from the original probe point
- $h_i(X) = (Hash(X) + i^2) \% \mathit{TableSize}$

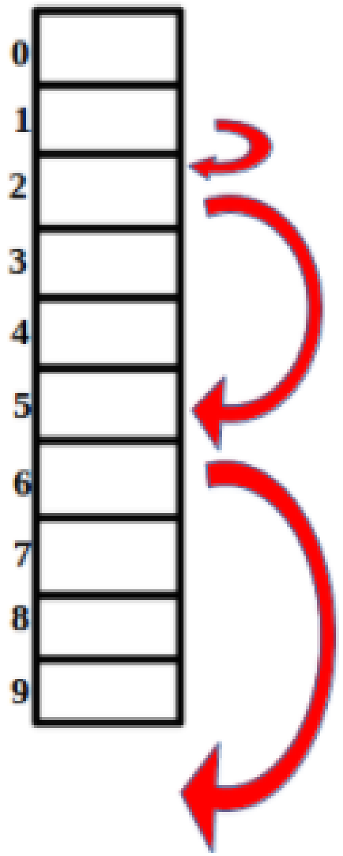
  h0(X) = Hash(X) % TableSize

  h1(X) = Hash(X) + 1 % TableSize

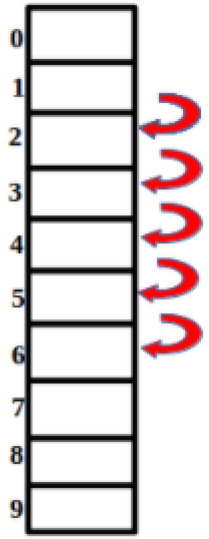
  h2(X) = Hash(X) + 4 % TableSize

  h3(X) = Hash(X) + 9 % TableSize

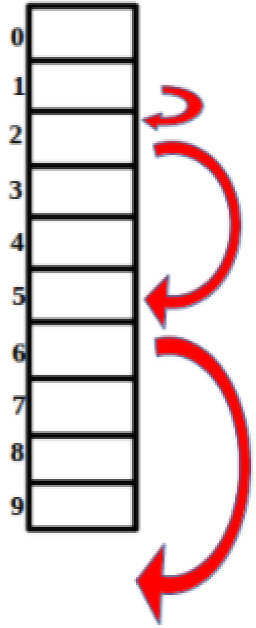
- Limitations
  - May not find a vacant cell; Table must be less than half full



Linear Probing and Quadratic Probing



Linear Probing



Quadratic Probing

A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

```
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

Problem with Quadratic Probing – Primary Clustering

- Works pretty well for an empty table and gets worse as the table fills up.
- If a bunch of elements hash to the same spot, they mess each other up.
- But, worse, if a bunch of elements hash to the same *area* of the table, they mess each other up! (Even though the hash function isn't producing lots of collisions!)
- This phenomenon is called primary clustering.

Double Hashing - Closed Hashing

- **Idea**: Spread out the search for an empty slot by using a second hash function
  - *No primary or secondary clustering*
- $h_i(X) = (Hash_1(X) + i * Hash_2(X)) \bmod \textit{TableSize}$
  for $i = 0, 1, 2, \ldots$
- Good choice of $Hash_2(X)$ can guarantee not getting "stuck" as long as $\lambda < 1$
  - Integer keys:
    $Hash_2(X) = R - (X \bmod R)$
    where R is a prime smaller than *TableSize*

Double Hashing

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(<u>k</u>) + i*g(<u>k</u>)) mod TableSize

where g is a second hash function

Double Hashing – Another Example

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
Probes	1	1	1	2	1	2

Rehashing

Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
- Cost of rehashing?

Go through old hash table, ignoring items marked deleted, Recompute hash value for each non-deleted key and put the item in new position in new table

Cannot just copy data from old table because the bigger table has a new hash function

Running time - O(N) – but infrequent.

But Not good for real-time safety critical applications

Methods to improve hashing Performance

- Use good hash functions
- Use larger table size
- Use good collision resolution methods