

# Open & Bucket Hashing

# *Problem Solving Activity on Friday!*

Binary Search Trees, Tries, Little Bit of Hashing

1. Manually building a BST to have a given shape given a bunch of numbers
2. Writing code to use BST's `insert` method to **code** a short function that builds a BST that has a given shape.
3. Understanding about how the shape of a Trie depends on the words contained inside of it
4. One short theoretical question about bucket hashing, which we will cover today.

## *Introduction*

**Hashing:** a method for storing and retrieving records from a database based on some attribute value of the records.

- Generate a *hopefully unique* key for each record
- Insertion, deletion, and search is based on the key value of the record

Careful implementation of hashing allows for constant time insertion, deletion, and search on average.

## *Hashing for Storage: The Model*

Imagine that you're searching for a book in a gigantic library.

- The books are not ordered meaningfully by title, author, ISBN, or anything
- Instead, to find a book, you use a "crystal ball" which tells you the shelf number where the book is located.



# *Introduction*

- Appropriate for
  - applications where all search is done by exact-match queries
- Not Appropriate for
  - applications where multiple records with the same key are permitted
  - answering range searches

Java Implementations: Hashtable, HashMap, HashSet

# *Hashing, Hash Systems, and Hash Tables*

Hashing creates some slightly overloaded terms. Some disambiguation:

- **Hashing** refers to the process of applying a hash function to a key.
- A **hash function** is a mathematical object that generates maps a key to an integer.
- A **hash table** is a data structure that stores records in an array. A record's position is determined by applying the hash function to the record's key.
- A **hash system** is a system that uses a hash table to store records & resolve collisions.

## Defining a Hash System

A hash system consists of a hash table and a hash function.

- A hash table  $T$  is an array of *slots*. Depending on the hash system, the slots may contain records or auxiliary data structures.
  - We use  $M$  to denote the number of slots in a hash table.
- A hash function  $h$  maps a key  $K$  to a slot in the hash table.
  - $h(K) = i$  refers to a slot in the hash table such that  $0 \leq i < M$ .

In a given hash system, a record with key  $K$  is stored at  $T[h(K)]$ .

## *The Crystal Ball: SUHA*

The **S**imple **U**niform **H**ashing **A**ssumption is a lie we tell ourselves in order to make analysis of hash-based storage easier.

Main idea:

- There exists (and we can use in constant time) a function that uniformly distributes inputs into slots
- Each item is equally likely to be placed into any slot

*Extraordinarily hard* to do exactly this given 1. the kind of data you and 2. the desired size of the table. But we can get close.



# Collisions

**Collision:** when two search keys are mapped by the hash function to the same slot in the hash table

Finding a record with key  $K$  in a database organized by hashing follows a two-step procedure:

- Compute the table location  $h(K)$
- Starting with slot  $h(K)$ , locate the record containing key  $K$  using (if necessary) a *collision resolution policy*

Under **SUHA**,  $P(\text{collision}) = \frac{n}{M}$ .

## Hashing & Collisions: An Example

First, we hash 🐱 to find its slot in the table.  $h(\text{🐱}) = 3$ .

We can place 🐱 in slot 3.

0	1	2	3	4	5	6	7	8	9
			🐱						

## Hashing & Collisions: An Example

Then, we hash 🐕 to find its slot in the table.  $h(\text{🐕}) = 8$ .

We can place 🐕 in slot 8.

0	1	2	3	4	5	6	7	8	9
			🐱					🐕	

## Hashing & Collisions: An Example

Then, we hash 🐟 to find its slot in the table.  $h(\text{🐟}) = 0$ .

We can place 🐟 in slot 0.

0	1	2	3	4	5	6	7	8	9
🐟			🐱					🐶	

## Hashing & Collisions: An Example

Then, we hash 🐘 to find its slot in the table.  $h(\text{🐘}) = 3$ .

We can try to place 🐘 in slot 3. But it's filled!

This is a **collision**—where would we put 🐘 so that:

- we can find it again?
- we don't overwrite any other records?

0	1	2	3	4	5	6	7	8	9
🐟			🐱					🐶	

## Hashing & Collisions: An Example

We can try to place 🐘 in slot 3. But it's filled!

**Could just put it in the next free slot...**

To find again, take  $h(\text{🐘}) = 3$ , see that it's filled with *something else*, then try going to the next slot.

0	1	2	3	4	5	6	7	8	9
🐟			🐱	🐘				🐶	

## Hashing & Collisions: An Example

We can try to place 🐘 in slot 3. But it's filled!



**Could "squeeze" it into the slot with the cat.**

To find again, take  $h(\text{🐘}) = 3$ , see that it's filled with multiple items, then search through those items.

0	1	2	3	4	5	6	7	8	9
🐟			[🐱, 🐘]					🐶	

# Collision Resolution Policies

Hashing and storing is simple, but collision resolution comes with a lot of tradeoffs!

- **Open hashing:** store multiple records in the same slot by using auxiliary data structures.
  -  Not all of the information is stored in the table itself
- **Closed hashing:** store one record per slot, and use a collision resolution policy to find a new slot for a record that collides with an existing record.
  -  All of the information is stored in the table itself

More (open) slots require more (unused) space, but allows for fewer collisions. Fewer slots requires less space, but may lead to more collisions, requiring more time to resolve.



## *Load Factor*

Used to decide when to rehash (resize) the hash table

$$\text{Load factor} = \frac{n}{M}$$

- $n$  = number of records in the hash table
- $M$  = the number of slots in the hash table

Resize and rehash the hash table when the load factor exceeds a certain threshold to keep it as low as possible.

Under **SUHA**,  $P(\text{collision}) = \alpha$ .

# *Open Hashing*

## *Open Hashing*

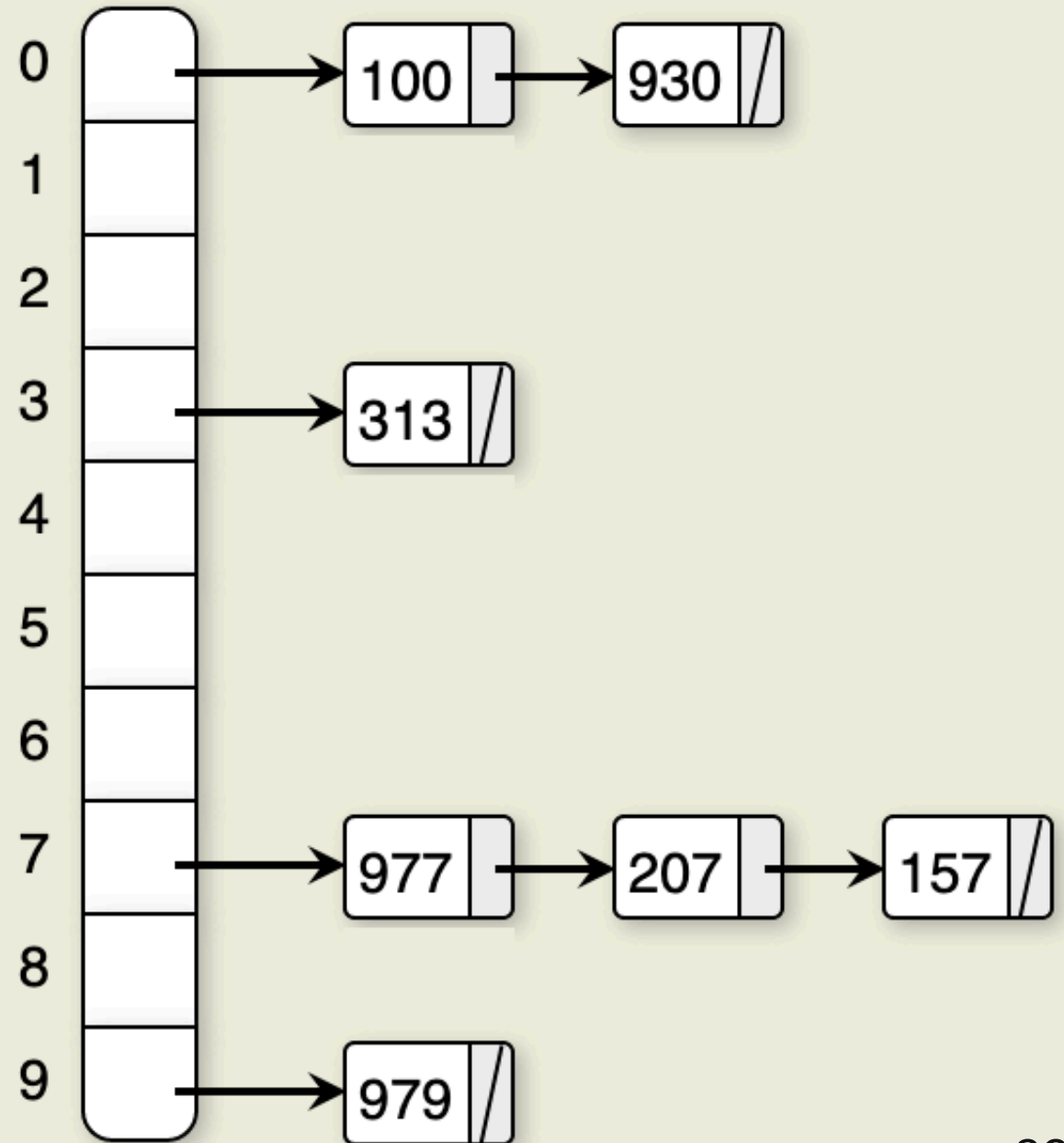
A hash system where multiple records might be associated with the same slot of a hash table.

- A linked list or other data structure is used to store the records in a slot

## Example

Open Hashing, using a simple mod (%) as hash function:

- $h(K) = K \% M$ , with  $M =$  the number of slots in the hash table.
- If we insert the keys: 157, 313, 930, 207, 979, 100, 977



# Open Hashing

- **Insertion:**
  - Compute the hash value of the key
  - Insert the record at the head of the linked list at the hash value
- **Searching:**
  - Compute the hash value of the key
  - Search the linked list at the hash value
- **Deletion:**
  - Compute the hash value of the key
  - Delete the record from the linked list at the hash value

Runtimes??

# Open Hashing

- **Insertion:**  $O\left(\frac{n}{M}\right)$  on average,  $O(n)$  in the worst case
  - Compute the hash value of the key
  - Insert the record at the head of the linked list at the hash value
- **Searching:**  $O\left(\frac{n}{M}\right)$  on average,  $O(n)$  in the worst case
  - Compute the hash value of the key
  - Search the linked list at the hash value
- **Deletion:**  $O\left(\frac{n}{M}\right)$  on average,  $O(n)$  in the worst case
  - Compute the hash value of the key
  - Delete the record from the linked list at the hash value

# Java HashMap

# HashMap

Hash function: bitwise AND (&) operator

```
public static int findSlot(Object toHash) {  
    slot = array_length & key.hashCode()  
}
```

Open Hashing System: collisions are added to a data structure stored in each slot.

- Data structure is a list when small, but a BST when above eight elements
- $O(\log n)$  in the worst case for insert/delete/contains



# *Closed Hashing*

## *Closed Hashing*

A hash system where all records are stored in slots inside the hash table.

Implementations:

- **Closed hashing with buckets**
- **Closed hashing with no buckets**

# *Closed Hashing with Buckets*

# Bucket Hashing

Slots of the hash table are grouped into **buckets**

- If the hash table has  $M$  slots and  $B$  buckets ( $M > B$ ), each bucket will consist of  $\frac{M}{B}$  slots.
- Additionally, the table will include an **overflow bucket**: the *bucket* into which a record is placed if the bucket containing the record's *home slot* is full
  - Overflow bucket is often considered to have infinite capacity—an `ArrayList`, perhaps

## ***Bucket Hashing: Insertion***

- Hash the key to determine which bucket should contain the record
- If the bucket is not full, insert the record in the first available slot
- If the bucket is full then store the record in the first available slot in the overflow bucket

*Worst case runtime? Average case, assuming SUHA?*

## *Bucket Hashing: Insertion*

*Assuming that duplicates aren't allowed...*

**Worst Case:** all elements end up in the same bucket, so we have to pass over every single element to verify we aren't adding it twice.  $O(n)$

**Average Case:** if no overflow, then  $n$  elements distributed over  $B$  buckets uniformly.  
→  $O\left(\frac{n}{B}\right)$  Once overflow, linear in length of the overflow bucket.

## ***Bucket Hashing: Searching***

Hash the key to determine which bucket should contain the record.

The records in this bucket are then searched.

- If the desired key value is not found and the bucket still has free slots, then the search is complete.
- If the bucket is full, then search the overflow bucket until the record is found or all records in the overflow bucket have been checked.

Exactly the same runtime as insertion:  $O\left(\frac{n}{B}\right)$  as long as there is space remaining; linear cost in overflow beyond that.

## *Bucket Hashing: Why?*

- In the limit, no better than searching through a linear structure. But!
- If you know how many elements you have and a hash function that approximates the **SUHA**, then you can use linear space for constant time operations.
- Can always resize & rehash if  $n$  grows—picking a suitably large new table size guarantees small cost per element.



## Reminder: Collisions = 😞

We hash 🐘 to find its slot in the table.  $h(\text{🐘}) = 3$ .

We can try to place 🐘 in slot 3. But it's filled!

This is a **collision**—where would we put 🐘 so that:

- we can find it again?
- we don't overwrite any other records?

0	1	2	3	4	5	6	7	8	9
🐟			🐱					🐶	

## *Collision Resolution Policy*

The process of finding the proper position in a hash table that contains the desired record.

- Used when searching/inserting if the hash function returns a slot that's occupied by a different item.
- Must admit a *reversible* procedure: the different slot we find needs to be traceable back to the original slot we tried to use.

## *Closed Hashing with No Bucket*

**Closed** → no auxiliary lists/trees allowed!

**No Bucket** → all records have to live in the table array!

Must have some collision resolution policy to figure out what happens when you try to use a slot that's occupied!

## *Picking a Hash Function*

Reminder: **SUHA** imagines that we have a perfectly uniform distribution of keys to slots. (This is more or less satisfied if we are hashing uniformly random integers with %)

So, for our examples:

```
int home = item.hashCode() % array_size
```

*Call it `home` instead of "slot" because this is no longer guaranteed to be the actual slot where we end up.*

## *Collision Resolution with Probing*

- Goal: find a free slot in the hash table when the home position for the record is already occupied
- Idea: Use a **probe function** to calculate some offset relative to the home position based on the *key* and the *probe count*.

# Closed Hashing Insertion w/ Collision Resolution

The **probe function**  $p(K, i)$  returns an offset from the original home position

1. Find home slot:  $\text{home} = h(K)$ 
  - $h$  is the hash function
  - $K$  is the key
2. If occupied, iteratively calculate positions the probe sequence until an empty slot is found with  $\text{pos} = (\text{home} + p(K, i)) \% M$ 
  - $p$  is our probe function
  - (Try home, then try probing with  $i = 1, i = 2, \dots$ )

## *Collision Resolution Policies*

- Linear probing
- Linear probing by steps
- Pseudo-random probing
- Quadratic probing
- Double hashing

## *Linear Probing*

Works by moving sequentially through the hash table from the `home`.

Probe function:  $p(k, i) = i$

Sequence of slots to try: `home`, `home + 1`, `home + 2`, `home + 3`, ...



**Reminder: Collisions =** 😞

Let's hash 13, 8, 103, 9, 33 using the  $\%M$  hash.

13 % 10 = 3, which is free

0	1	2	3	4	5	6	7	8	9

## Reminder: Collisions = 😞

Let's hash 13, 8, 103, 9, 33 using the  $\%M$  hash.

$8 \% 10 = 8$ , which is free

0	1	2	3	4	5	6	7	8	9
			13						

**Reminder: Collisions =** 😞

Let's hash 13, 8, 103, 9, 33 using the  $\%M$  hash.

$103 \% 10 = 3$ , which is full

$3 + p(k, 1) = 3 + 1 = 4$ , which is free

0	1	2	3	4	5	6	7	8	9
			13					8	

## Reminder: Collisions = 😞

Let's hash 13, 8, 103, 9, 33 using the  $\%M$  hash.

$9 \% 10 = 9$ , which is free

0	1	2	3	4	5	6	7	8	9
			13	103				8	

## Reminder: Collisions = 😞

Let's hash 13, 8, 103, 9, 33 using the  $\%M$  hash.

$33 \% 10 = 3$ , which is full

$3 + p(k, 1) = 3 + 1 = 4$ , which is full

$3 + p(k, 2) = 3 + 2 = 5$ , which is free

0	1	2	3	4	5	6	7	8	9
			13	103				8	9

## Reminder: Collisions = 😞

Let's check if 33 is present...

$33 \% 10 = 3$ , that slot does not have the target, but next slot in probe sequence might have it...

$3 + p(k, 1) = 3 + 1 = 4$ , that slot does not have the target, but next slot in probe sequence might have it...

$3 + p(k, 2) = 3 + 2 = 5$ , which is the target! DONE, 33 is present.

0	1	2	3	4	5	6	7	8	9
			13	103	33			8	9

## Reminder: Collisions = 😞

Let's check if 18 is present...

$18 \% 10 = 8$ , that slot does not have the target, but next slot in probe sequence might have it...

$8 + p(k, 1) = 8 + 1 = 9$ , that slot does not have the target, but next slot in probe sequence might have it...

$8 + p(k, 2) = 8 + 2 \rightarrow 0$ , which is empty! DONE, 18 is **not** present.

0	1	2	3	4	5	6	7	8	9
			13	103	33			8	9

## Observations

- Hashing repeatedly to the same home slot leads to long chains of occupied slots.
- Can't finish insert/search operations until we find an empty slot in a probe sequence.
- Slots filled by hashing to unrelated home slots can interfere in the insertion/search for other elements.
  - putting 9 in its home slot made searching for a key that hashed to home slot 8 harder!



## Primary Clustering

- The tendency in certain collision resolution methods to create clustering (groups of occupied slots) in sections of the hash table
- Happens when a group of keys follow the same probe sequence during collision resolution
- Primary clustering leads to empty slots in the table to not have an equal probability of receiving the next record inserted
  - Always want  $P(\text{slot} = i) = \frac{1}{M-n}$  for all free slots  $i$ .

## Primary Clustering

Assuming SUHA, what's the probability that our next element ends up in slot 6?

0	1	2	3	4	5	6	7	8	9
			13	103	33			8	9

## Primary Clustering

Assuming SUHA, what's the probability that our next element ends up in slot 6?

Home slots of 3, 4, 5, and 6 all lead there!  $\Rightarrow P(\text{slot} = 6) = \frac{4}{10}$

0	1	2	3	4	5	6	7	8	9
			13	103	33			8	9

## *Primary Clustering*

- Linear probing leads to primary clustering in a big way!
- Linear probing is one of the worst collision resolution methods.

## Linear Probing by Steps

**Goal:** avoid primary clustering / improve linear probing

**Idea:** skip slots by some constant  $c$  other than 1

**Probe function:**  $p(k, i) = ci$

- $c$  must be relatively prime to  $M$  to generate a linear probing sequence that visits all slots in the table
- i.e.  $c = 2$  gets us  $[0, 2, 4, 6, 8]$  but  $c = 7$  gets us  $[0, 7, 4, 1, 8, 5, 2, 9, 6, 3]$  starting from 0.

## Linear Probing by Steps

Hash 10, then 40 using  $p(k, i) = 7i$

10 goes in slot 0

0	1	2	3	4	5	6	7	8	9
			13	103	33			8	9

## Linear Probing by Steps

Hash 10, then 40 using  $p(k, i) = 7i$

40 goes in slot 0, which is full. So try 7, which is empty!

0	1	2	3	4	5	6	7	8	9
10			13	103	33			8	9

## Linear Probing by Steps

Hash 10, then 40 using  $p(k, i) = 7i$

40 goes in slot 0, which is full. So try 7, which is empty!

0	1	2	3	4	5	6	7	8	9
10			13	103	33		40	8	9



## Linear Probing by Steps

"Is 37 present?"

Check slot 7, which is full. Check slot  $7 + 7 \% 10 = 4$ , which is full. Check  $4 + 7 \% 10 = 1$ , which is empty. 37 is not present!

0	1	2	3	4	5	6	7	8	9
10			13	103	33		40	8	9

## *Are We Solving Primary Clustering?*

1. What is the sequence of slots probed when hashing a record with home slot of  $0$  with  $p(k, i) = 3i$  and  $M = 10$ ?
2. What is the sequence of slots probed when hashing a record with home slot of  $3$  with  $p(k, i) = 3i$  and  $M = 10$ ?

## Are We Solving Primary Clustering?

1. What is the sequence of slots probed when hashing a record with home slot of  $0$  with  $p(k, i) = 3i$  and  $M = 10$ ?
  - i. **0, 3, 6, 9, 2, 5, 8, 1, 4, 7**
2. What is the sequence of slots probed when hashing a record with home slot of  $3$  with  $p(k, i) = 3i$  and  $M = 10$ ?
  - i. **3, 6, 9, 2, 5, 8, 1, 4, 7, 0**

## ***Linear Probing With Steps: No Good!***

It is true that records with adjacent home positions will not follow the same probe sequence. BUT! Records offset by  $c$  *will* follow the same probe sequence.

Clusters will not be formed of adjacent cells, but primary clustering still happens!

## Quadratic Probing

**Goal:** avoid primary clustering by creating probe sequences that have little overlap

**Idea:** skip slots by increasingly large step sizes

**Probe function:**  $p(k, i) = c_1i^2 + c_2i + c_3$  (a quadratic function)

- Kind of hard mathematically to choose constants such that the sequence covers the table
- Solutions:
  - If  $T$  is a power of 2, then use  $p(k, i) = \frac{i^2+i}{2}$
  - If  $T$  is prime, use  $p(k, i) = i^2$

## Overlapping Probe Sequences

Assume  $T = 10$  and  $p(k, i) = i^2$

1. What's the probe sequence for hashing with home slot of 5?
2. What's the probe sequence for hashing with home slot of 6?

## Overlapping Probe Sequences

Assume  $T = 10$  and  $p(k, i) = i^2$

1. What's the probe sequence for hashing with home slot of 5?

i. **5, 6, 9, 4, 1, 0, 1, 4, 9, 6, 5, ...**

2. What's the probe sequence for hashing with home slot of 6?

i. **6, 7, 0, 5, 2, 1, 2, 5, 0, 7, 6, ...**

*We don't hit all of the possible locations, but at least the sequences aren't identical!*

## *Random Probing(?)*

**Goal:** avoid primary clustering by creating probe sequences that have little overlap

**Idea:** just choose a random new location

**Probe function:** `(int) (Math.random() * M)`

- Bare minimum clustering...
- ... but impossible to retrieve an element once it's been added. (If there's a collision, how would we know that the next probe that we get goes to the same location?)



## *Pseudo-Random Probing(?)*

**Goal:** avoid primary clustering by creating probe sequences that have little overlap

**Idea:** Generate an arbitrary permutation of  $[1, M-1]$  and use those as probe offsets

**Probe function:**  $p(k, i) = \text{permutation}[i]$

- Positive benefits of random clustering, no issues with "reproducibility"

## Overlapping Probe Sequences

Assume permutation = {3, 8, 1, 4, 2, 9, 0, 5, 7, 8, 6}

1. What's the probe sequence for hashing with home slot of 5?
2. What's the probe sequence for hashing with home slot of 6?

## Overlapping Probe Sequences

Assume permutation = {3, 8, 1, 4, 2, 9, 0, 5, 7, 8, 6}

1. What's the probe sequence for hashing with home slot of 5?

i. **8, 3, 6, 9, 7, 4, 5, 0, 2, 3, 1**

2. What's the probe sequence for hashing with home slot of 6?

i. **9, 4, 7, 0, 8, 5, 6, 1, 3, 4, 2**

*Easy to hit all locations, not much sequence overlap!*

## *Eliminating Primary Clustering!*

Pseudo-random and Quadratic Hashing both successfully eliminate primary clustering!

- Overlap in probe sequence is, on average, pretty low

But:

- If  $\text{home}(k_1) == \text{home}(k_2)$ , two keys will always follow the same probe sequence.
- Probe sequences are entirely functions of the home slot— $p(k, i)$  ignores  $k$  always.
- Both lead to **secondary clustering**—if there's a cluster in home slots, there will be clustering down the sequence, too.

## *Eliminating Secondary Clustering?*

**Goal:** Make probe sequences diverge with the same home key

**Idea:** Hash again!  $p(i, k) = ih_2(k)$

## *Number Theory*

Important that:

- outputs of the second hash remain in the range  $[1, M-1]$
- impossible to output  $0$  or  $M$ —the "offset" for the probe would be no offset at all

*The first hash for the home slot is easy to think about, the second one takes a little more careful planning.*

## *Prime Table Double Hashing*

Use a prime value of  $M$ !

$$h_1(k) = k \% M$$

$$h_2(k) = 1 + (k \% (M - 1))$$

$$p(k, i) = i * h_2(k)$$

# Quick Analysis

