CS 2100: Data Structures & Algorithms 1

Hash Tables

Open Addressing; Analysis on Hashing

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

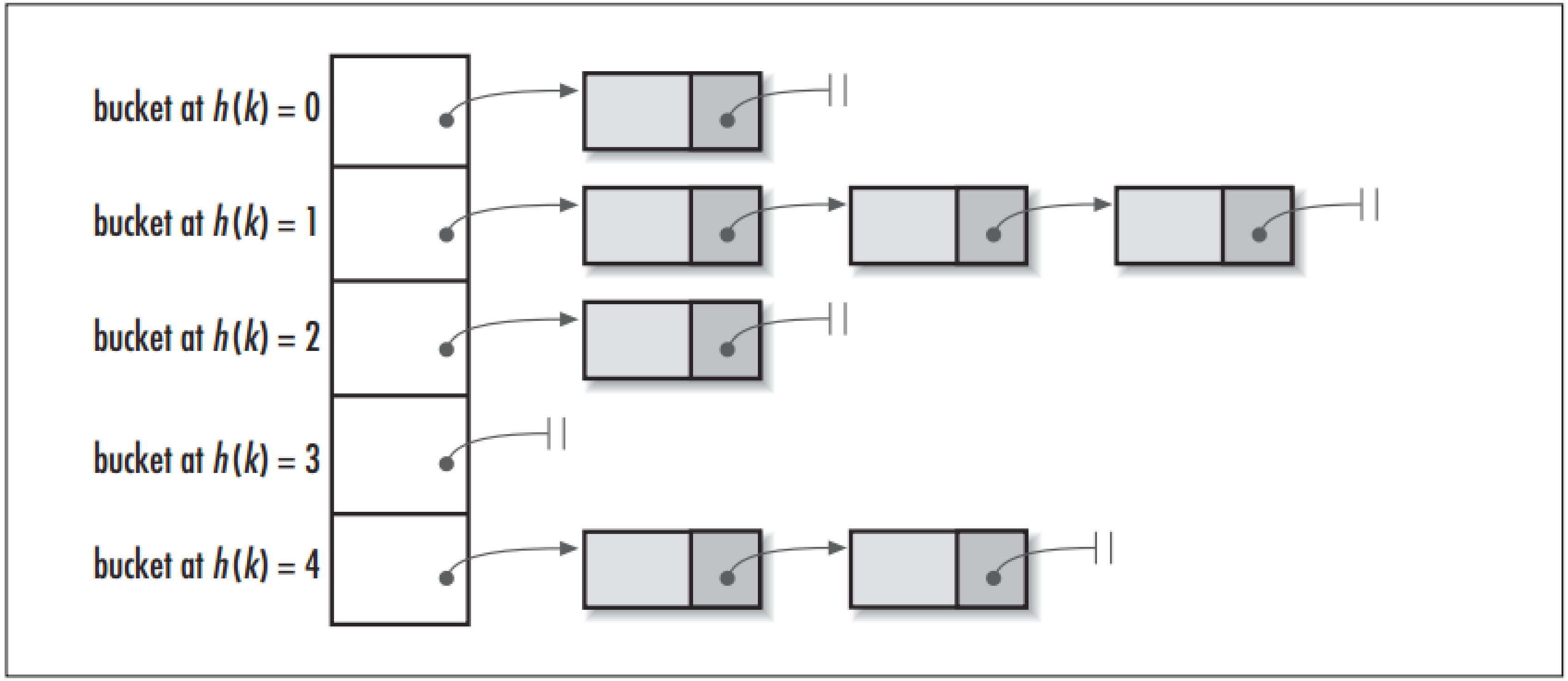
- The University updated the mask policy. As per my Request on Mar 28, 2022 (see Collab), I would greatly appreciate if you would do me a kind favor by **continuing to wear your masks** in CS 2100 (Ridley G008). I know it is a lot to ask, and it is **voluntary**, but I appreciate your understanding.
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

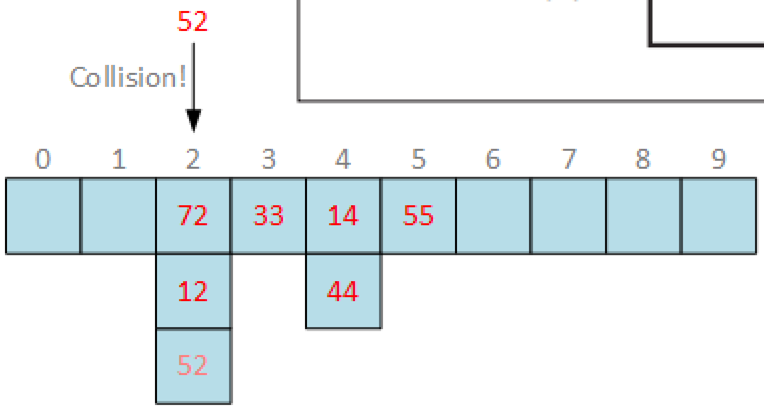
Separate Chaining

A Collision Resolution Technique

We Spoke About…

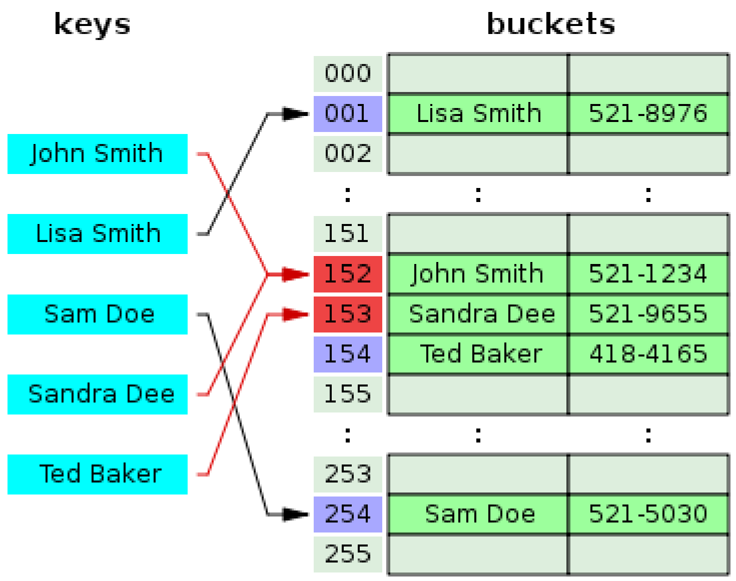
Collision Resolution: Separate Chaining





Open Addressing

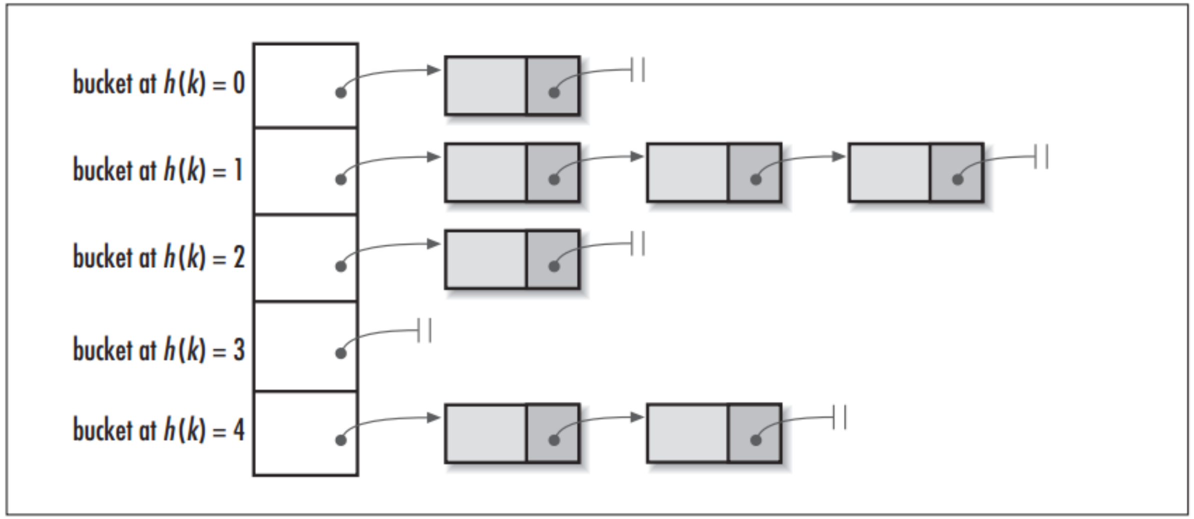
Another Collision Resolution Technique



000		
001	Lisa Smith	521-8976
002		
:	:	:
151		
152	John Smith	521-1234
153	Sandra Dee	521-9655
154	Ted Baker	418-4165
155		
:	:	:
253		
254	Sam Doe	521-5030
255		

Saving Memory

- Can we avoid the overhead of all those linked lists?
  - *Separate Chaining is a great collision resolution technique, but it takes up a lot of* ***extra space!***



Three Types of Probing Strategies

- **Open Addressing** is a Collision Resolution technique that is sometimes called by its specific type, for example "Linear Probing"

- **There are three types of probing strategies:**
  - Linear
  - Quadratic
  - Double hashing
- The general idea with all of them is that, if a spot is occupied, to '**probe**', or **try**, other spots in the table to use
  - How we determine where else to probe depends on which strategy we are using

Linear Probing Collision Resolution Technique

- If faced with a collision situation, the **linear probing** technique will look into subsequent slots until the first free space is found
  - When probing for an empty slot, we take one “step” at a time, it is called **linear probing**

- Check spots in this order:
  - `hash(k), hash(k)+1, hash(k)+2`, etc.

- `hash(k) = 3k+7`
  - Which is then mod'ed by the table size (10)
  - Result: `hash(k) = (3k+7) mod 10`

- Insert: 4, 27, 37, 14, 21
  - hash(k) values: 19, 88, 118, 49, 70, respectively

Linear Probing Collision Resolution Technique

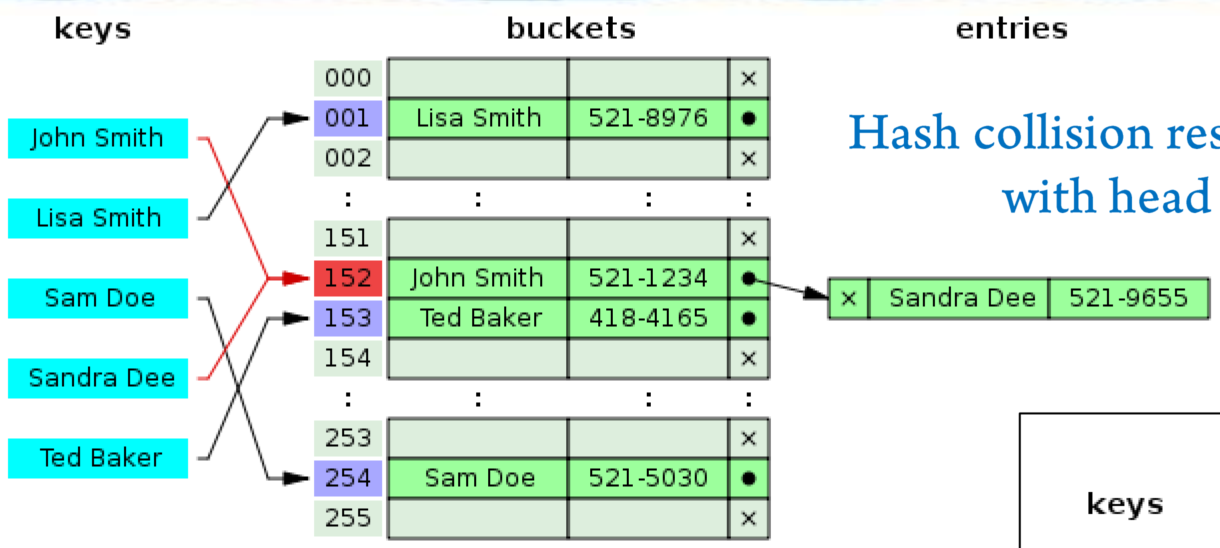
- With all **open addressing schemes**, we examine ('**probe**') the cells in the order:
  - `p`$_0$`(k), p`$_1$`(k), p`$_2$`(k), ...`
  - where: `p`$_i$`(k) = (hash(k) + `**`f(i)`**`)` **`mod`** `table_size`

- With linear probing, **`f(i)`** `= i`
  - After searching spot **hash(k)** in the array, look in:
    - `hash(k) + 1`
    - `hash(k) + 2`
    - `hash(k) + 3`
    - etc.

Problems With Linear Probing

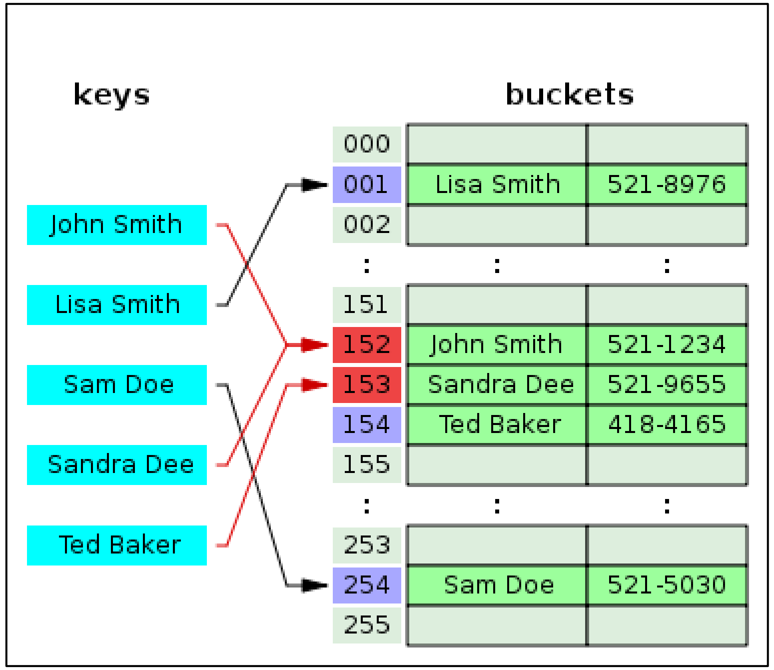
- **Primary clustering**
  - Large blocks of occupied cells
  - As table fills, increased number of attempts required to solve collision
    - And thus, **slower** lookup times
  - "Holes" when an element is removed
    - We'll see how to solve this later
  - When to stop looking?
    - Not always clear.
    - Continue until you find the element (linear search the remainder of the list!)
    - Continue until you get to the end of the structure (when you do not find the target)

Example of Hash Collision Resolution



Hash collision resolved by separate chaining with head records in the bucket array

Hash collision resolved by open addressing with linear probing (interval=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith"



Quadratic Probing Collision Resolution Technique

- With all **open addressing schemes**, we examine ('**probe**') the cells in the order:
  - $p_0(k), p_1(k), p_2(k), \ldots$
  - where: $p_i(k)$ = `(hash(k) + f(i)) mod table_size`
- With quadratic probing, $f(i) = i^2$
  - After searching spot **hash(k)** in the array, look in:
    - $\text{hash}(k) + 1^2 = \text{hash}(k) + 1$
    - $\text{hash}(k) + 2^2 = \text{hash}(k) + 4$
    - $\text{hash}(k) + 3^2 = \text{hash}(k) + 9$
    - etc.

Spreads things out
a bit more
(no large clusters)

- Remember, the hash function could be: `hash(k) = 3k+7`
  - Insert: 4, 27, 37, 14, 21
    - hash(k) values: 19, 88, 118, 49, 70, respectively

Double Hashing Collision Resolution Technique

- With all **open addressing schemes**, we examine ('**probe**') the cells in the order:
  - `p0(k), p1(k), p2(k), ...`
  - where: `pi(k) = (hash(k) + f(i)) mod table_size`
- With double hashing, `f(i) = i * hash2(k)`
  - Which means we have to define a secondary hash function!
  - After searching spot **hash(k)** in the array, look in:
    - `hash(k) + 1 * hash2(k)`
    - `hash(k) + 2 * hash2(k)`
    - `hash(k) + 3 * hash2(k)`
    - etc.

Combine with a ***secondary*** hash function!

Double Hashing Collision Resolution Technique

- Check spots in this order:
  - hash(k)
  - hash(k) + 1 * hash$_2$(k)
  - hash(k) + 2 * hash$_2$(k)
  - hash(k) + 3 * hash$_2$(k)
- hash(k) = k
  - Result: hash(k) = k mod 10
- hash$_2$(k) = 7 – (k mod 7)
- Insert: 89, 18, 49, 58, 69, 60

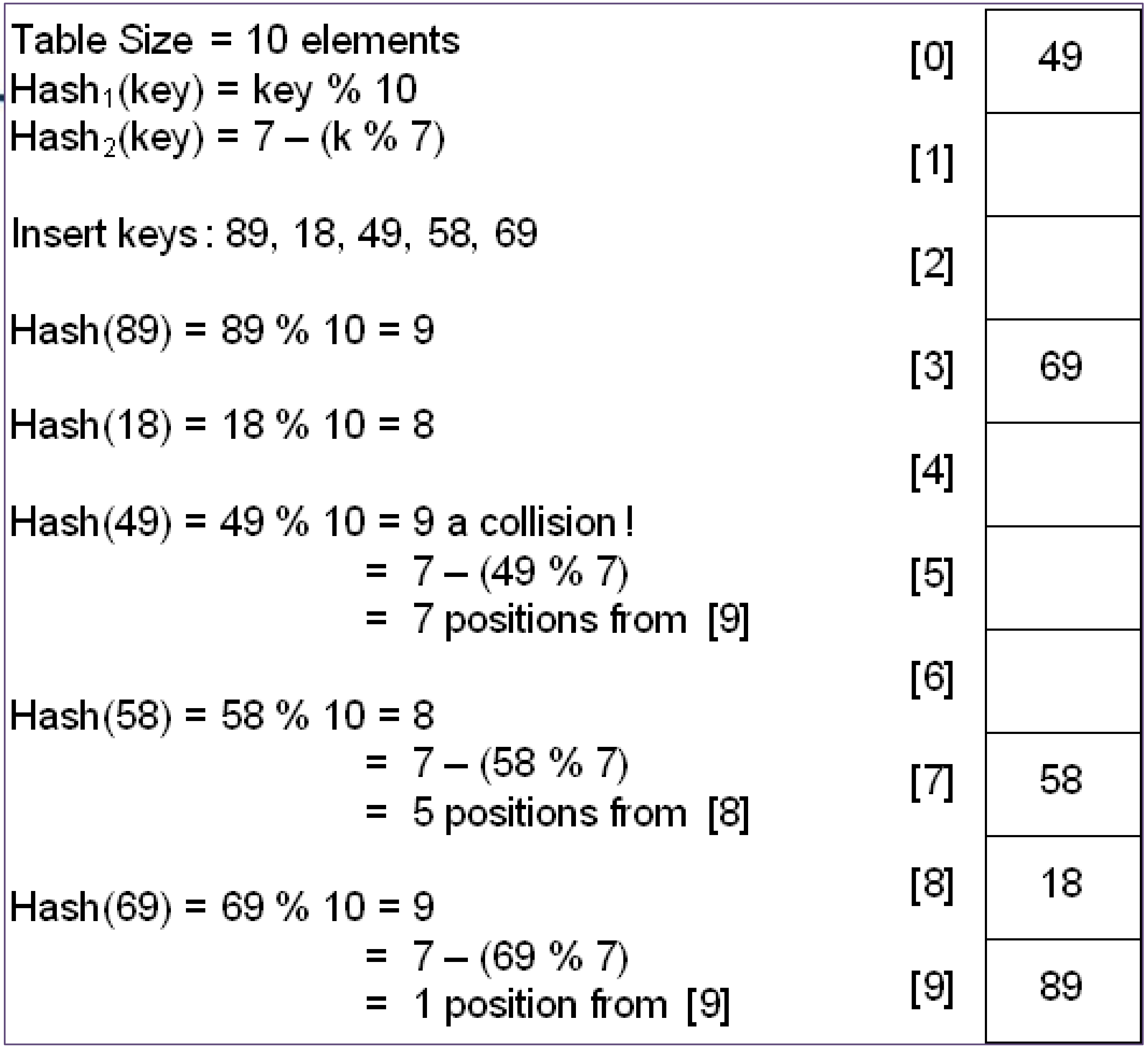


Table Size = 10 elements
Hash$_1$(key) = key % 10
Hash$_2$(key) = 7 – (k % 7)

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Double Hashing Collision Resolution Technique

- Check spots in this order:
  - hash(k)
  - hash(k) + 1 * hash$_2$(k)
  - hash(k) + 2 * hash$_2$(k)
  - hash(k) + 3 * hash$_2$(k)
- hash(k) = k
  - Result: hash(k) = k mod 10
- hash$_2$(k) = 7 – (k mod 7)
- Insert: 89, 18, 49, 58, 69, 60

A prime number

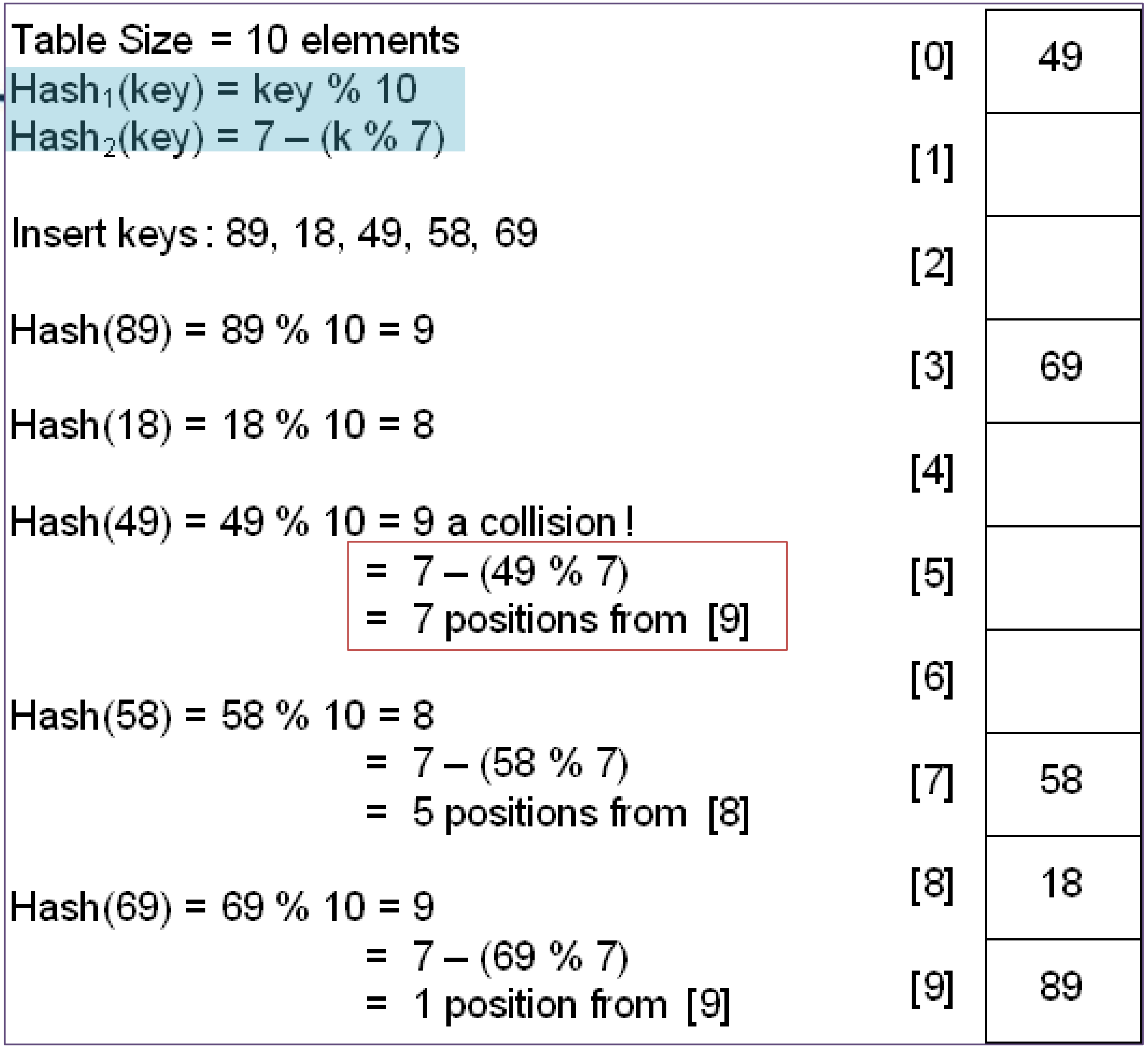


Table Size = 10 elements

Hash_1(key) = key % 10

Hash_2(key) = 7 – (k % 7)

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Double Hashing THRASHING

- `hash(k) = k mod 10`
  - Same as the previous slide
  - Result: `hash(k) = k mod 10`
- $hash_2(k) = (k \bmod 5) + 1$

Insert: 10, 12, 14, 16, 18, 36

- 10 mod 10 = 0
- 12 mod 10 = 2
- 14 mod 10 = 4
- 16 mod 10 = 6
- 18 mod 10 = 8
- 36 mod 10 = 6
  - Collision!
  - (36 mod 5)+1 = 2
    2 positions from 6 = 8
  - Collision!
  - 2 * ((36 mod 5)+1) = 4
    4 positions from 8 = 2
  - Collision!
  - . . .
  - ***It never lands on an empty slot; the hash function keeps thrashing in attempt to find a slot to place the key.***

Table Size Must Be Prime!

- The table size must always be a prime number
  - Thrashing will only occur when the double hash value is a *factor* of the table size
  - The only factors of a prime number *p* are 1 and *p*
  - It will provide better distribution of the hash keys into the table
    - Less clustering, etc.
- A prime number table size does not remove the need for a **good hash function!**

Miscellaneous

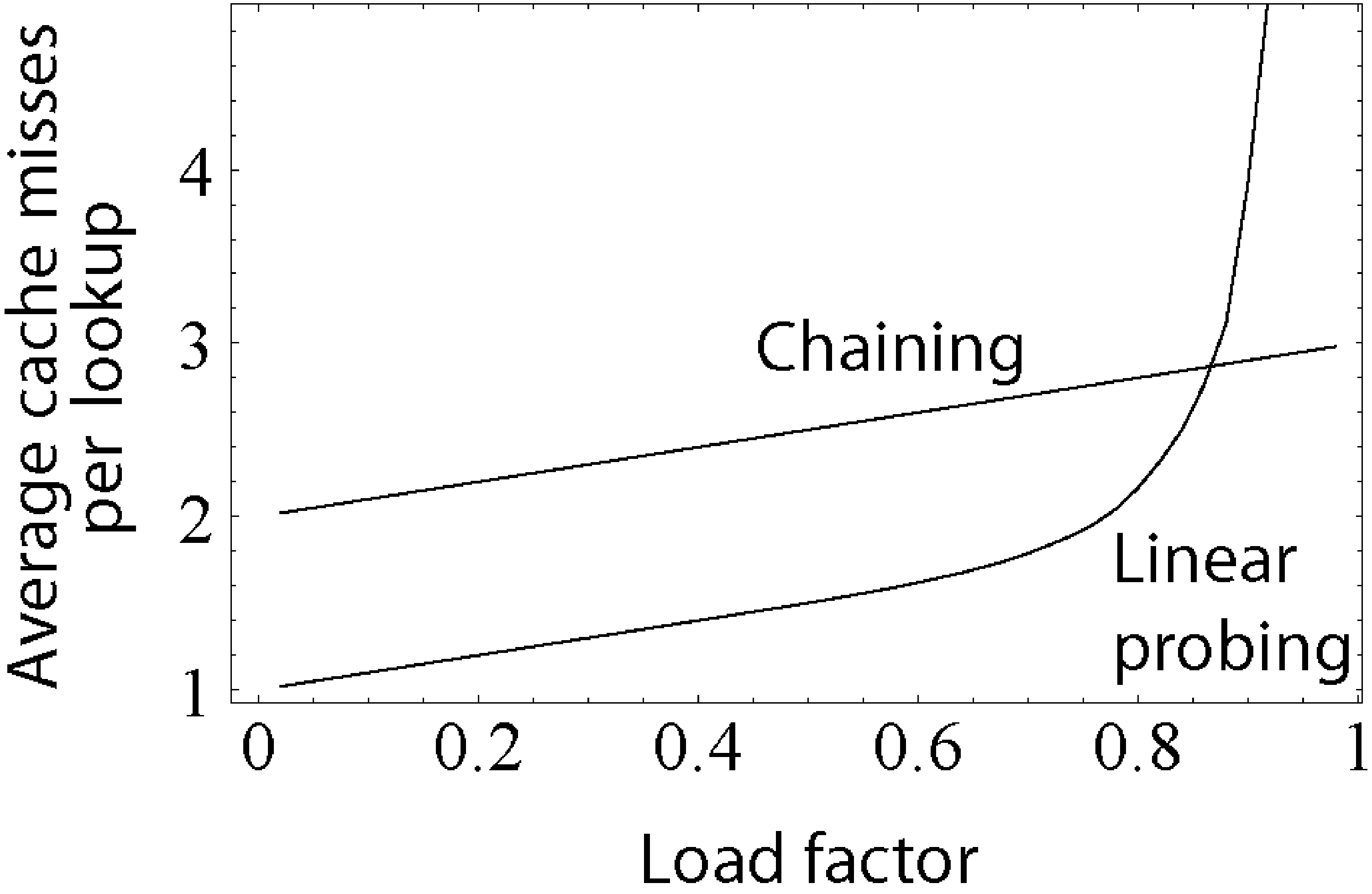
Rehashing

- **Problem**: when the table gets too full, running time for operations increases
- **Solution**: create a bigger table and hash all the items from the original table into the new table
  - The position in a table is dependent on the **table size**, which means we have to **rehash** each value
  - This means we have to **re-compute** the hash value for each element and **insert** it into the new table!

Rehashing

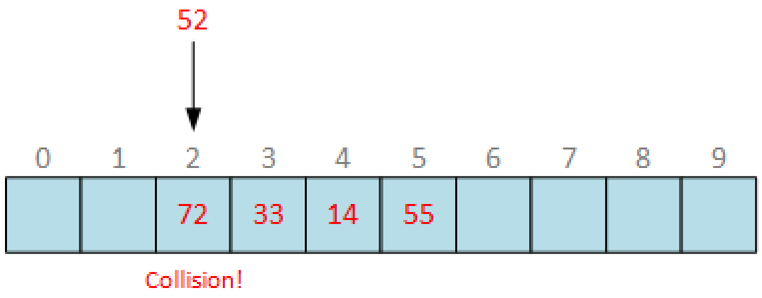
- **When to rehash?**
  - When half full ($\lambda = 0.5$)
  - When mostly full ($\lambda = 0.75$)
    - Java's **hashtable** does this by default
  - When an insertion fails
- Cost of rehashing
  - We have to do *n* inserts so worst case **$\Theta(n^2)$** operation!

Chaining vs. Linear Probing

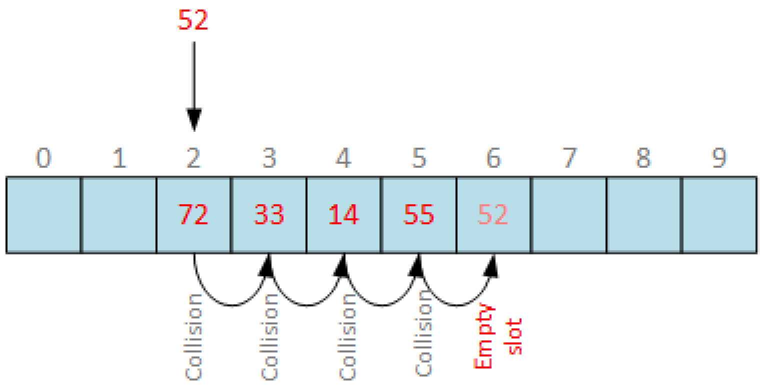


Removing an Element

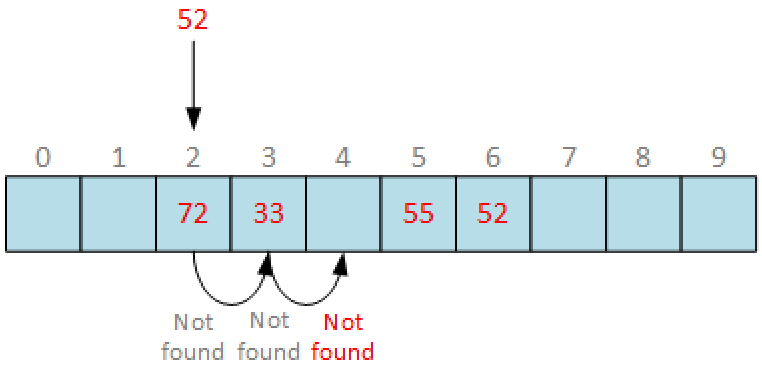
- How to handle this?
- You could:
  - Rehash upon each delete, which is very **expensive**
  - Put in a 'placeholder' or **'sentinel'** value (often assigned to “null”)
    - But the table gets filled with these rather fast
    - Perhaps rehashing after a certain number of deletes
  - Disallow deletes entirely; *Not recommended*
- Hash tables are not an ideal data structure if you need to perform a lot of deletions



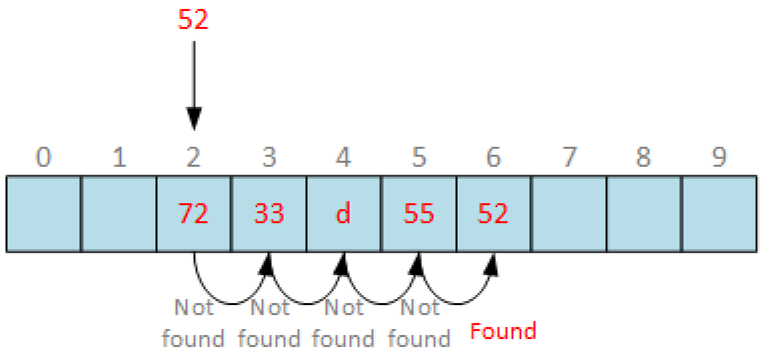
Oh dear… collision



Linear probe to find an empty slot



Deletion Anomaly (delete 14)
Searching problem…
Couldn't find 52…
but 52 is there!



With the **sentinel** value
('d' pictorially for
deleted element),
now 52 *can be found!*

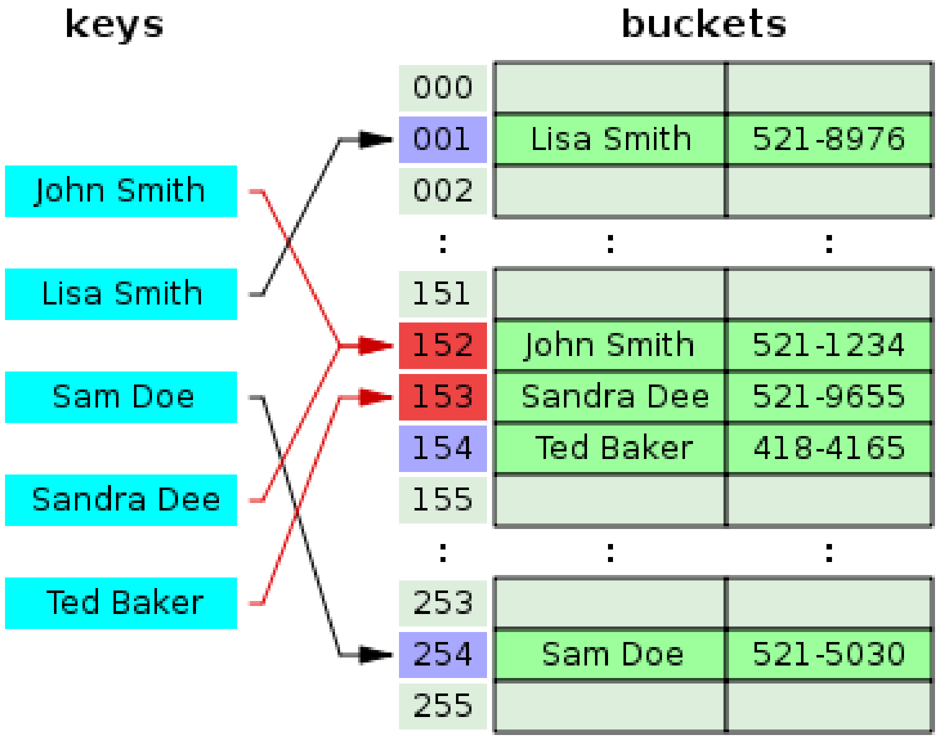
Sentinel Value

- When **deleting** an element, **replace** it with a “**sentinel**” element (because when inserting an item, you may have stepped over this element to place the item)
  - Will lead to deletion anomaly if you delete an element without replacing it with a sentinel.
- When **inserting**…
  - Can **place an element in** this sentinel position (as it represents an empty slot)
- When **searching**…
  - Can **continue probing** once you hit the sentinel position (as it represents a slot that was occupied previously)
    - This prevents you from prematurely stopping the search when the item you are searching for is further along

Deletion Anomaly

1. Add John Smith
2. Add Lisa Smith
3. Add Sandra Dee
    - Linear Probe to add
4. Add Ted Baker
    - Linear Probe to add
5. REMOVE John Smith
6. SEARCH for Sandra Dee

*Need to include **sentinel value** when deleting John Smith in order to successfully find Sandra Dee!*



Other Uses Of Hashing

- Storing passwords: increases security.
- Security of downloads (SHA-2)
- Cryptocurrencies (Hash functions used to verify transactions)