CSE 332 Autumn 2023
Lecture 12: Hashing

Nathan Brunelle

http://www.cs.uw.edu/332

Next topic: Hash Tables

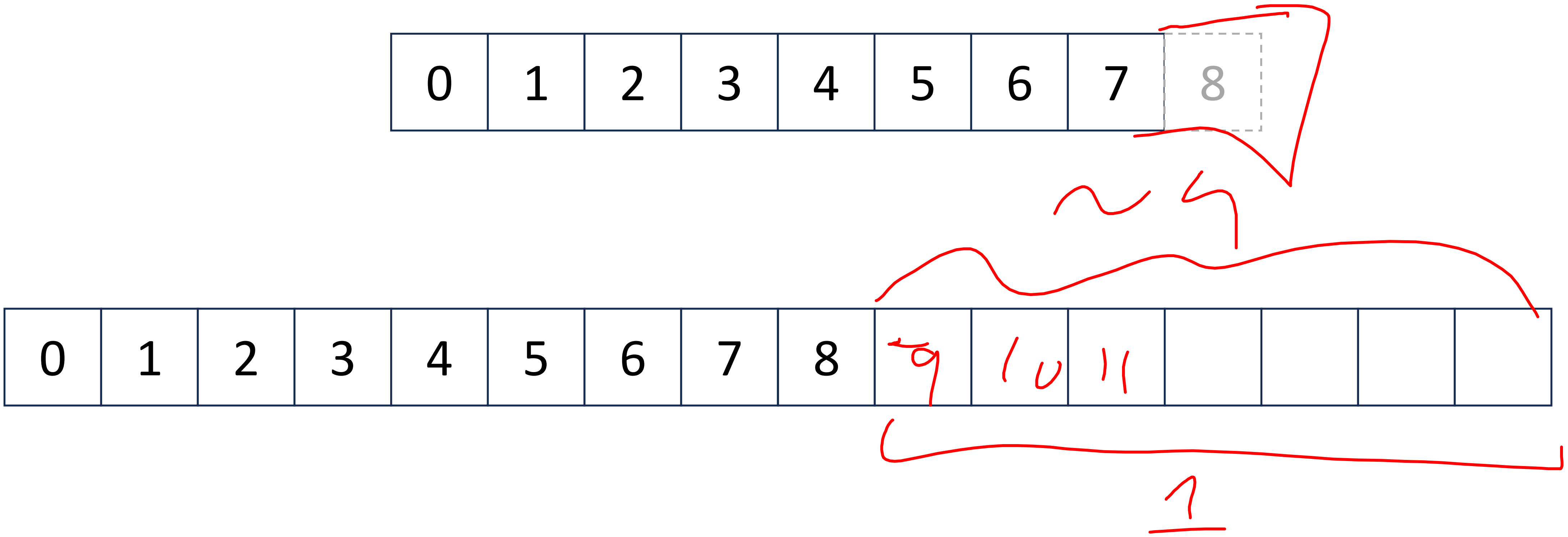
Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Hash Table (Worst case)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Hash Table (Average)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Two Different ideas of "Average"

- Expected Time
  - The expected number of operations a randomly-chosen input uses
  - Assumed randomness from somewhere
    - Most simply: from the input
    - Preferably: from the algorithm/data structure itself
  - $f(n)$ = sum of the running times for each input of size n divided by the number of inputs of size n
- Amortized Time
  - The long-term average per-execution cost (in the worst case)
  - Rather than look at the worst case of one execution, look at the total worst case of a sequential chain of many executions
    - Why? The worst case may be guaranteed to be rare
  - $f(n)$ = the sum of the running times from a sequence of n sequential calls to the function divided by n

Amortized Example

- ArrayList Insert:
  - Worst case: $\Theta(n)$



Amortized Example

- ArrayList Insert:
  - First 8 inserts: 1 operation each
  - 9^{th} insert: 9 operations
  - Next 7 inserts: 1 operation each
  - 17^{th} insert: 17 operations
  - Next 15 inserts: 1 operation each
  - …

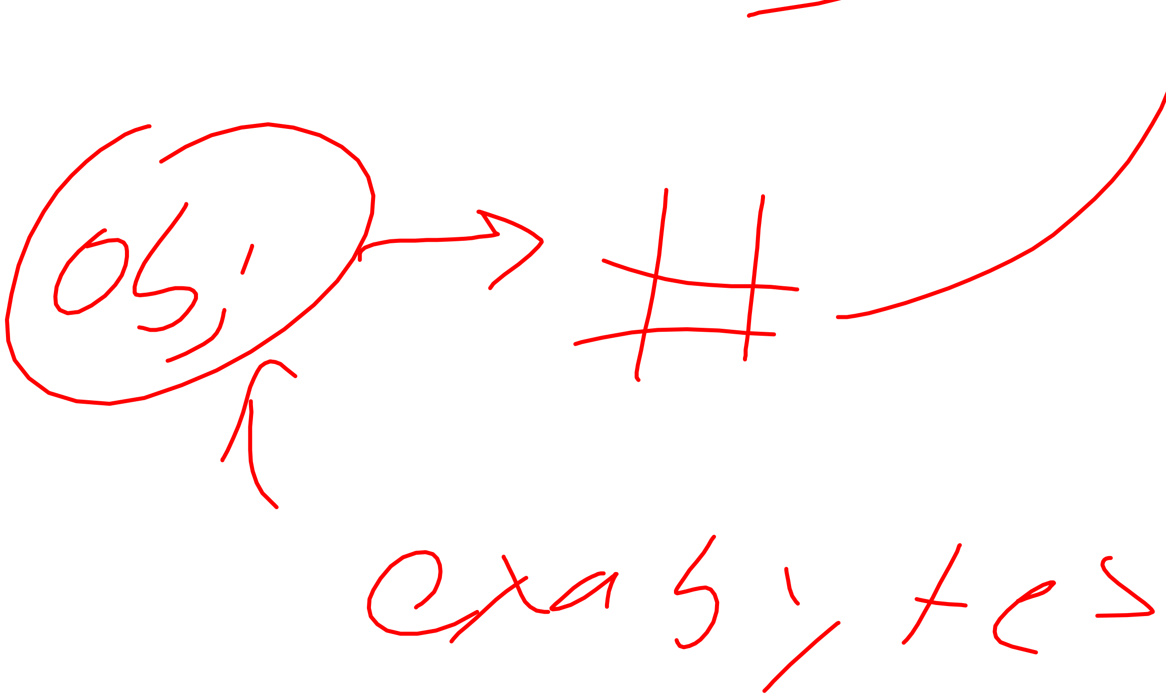
Do x operations with cost 1
Do 1 operation with cost x
Do x operations with cost 1
Do 1 operation with cost $2x$
Do $2x$ operations with cost 1
Do 1 operation with cost $4x$
Do $4x$ operations with cost 1
Do 1 operation with cost $8x$
…
Amortized: each operation cost 2 operations
$\Theta(1)$

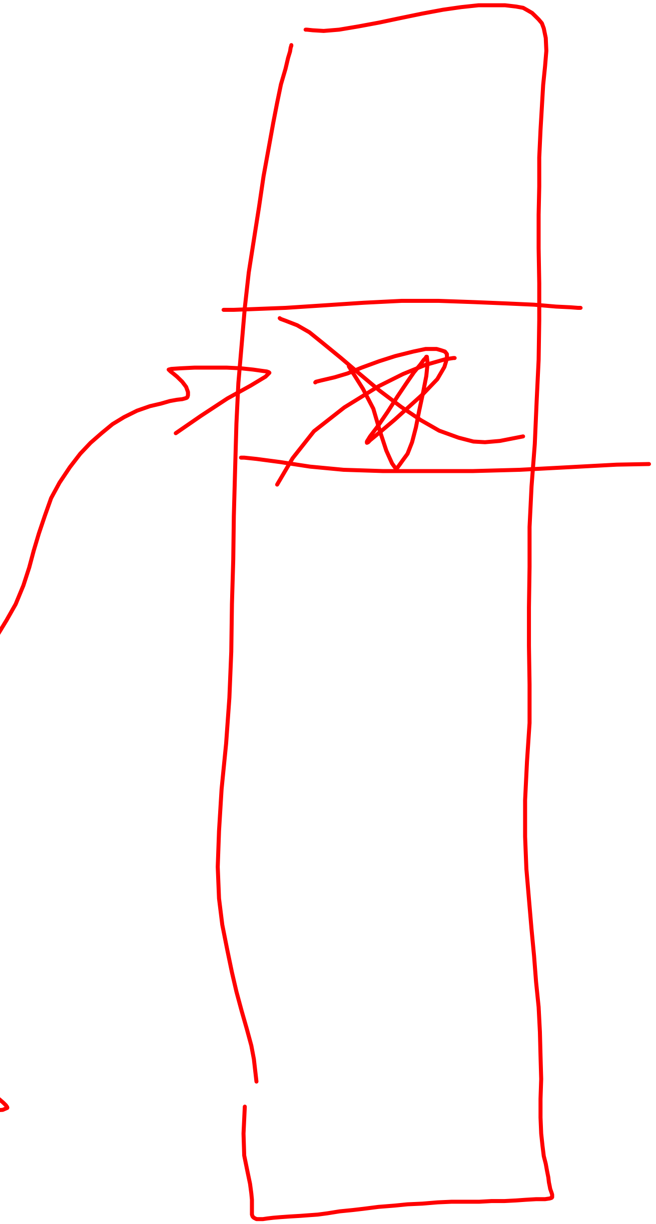
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8							

Hash Tables

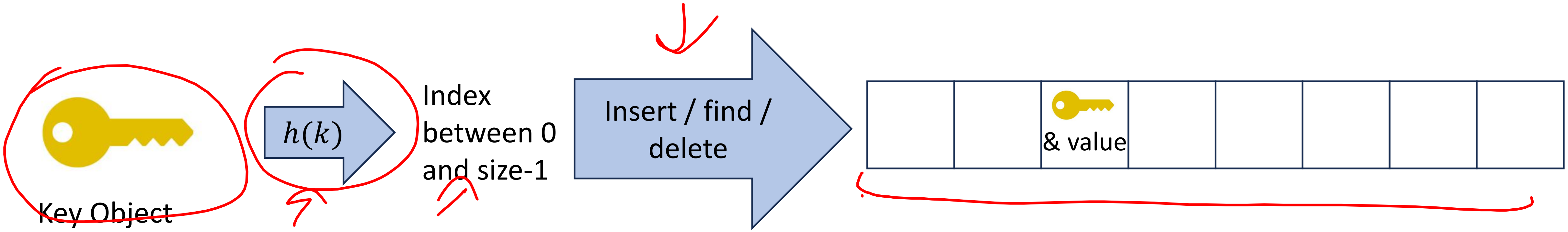
- Motivation:
    - Why not just have a gigantic array?





Hash Tables

- Idea:
    - Have a small array to store information
    - Use a **hash function** to convert the key into an index
        - Hash function should "scatter" the keys, behave as if it randomly assigned keys to indices
    - Store key at the index given by the hash function
    - Do something if two keys map to the same place (should be very rare)
        - Collision resolution



Example

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

10

- Key: Phone Number
- Value: People
- Table size: 10
- $h(phone)$ = number as an integer % 10
- $h(8675309) = 9$

8675319

8675301

What Influences Running time?

- How “spread out” our input keys are
  - How much do keys repeat
- Hash the function itself will take time
- Size of the table relative to the number things inserted
- How well our hash function scatters the keys
- What do we do when two things hash to the same spot

Properties of a "Good" Hash

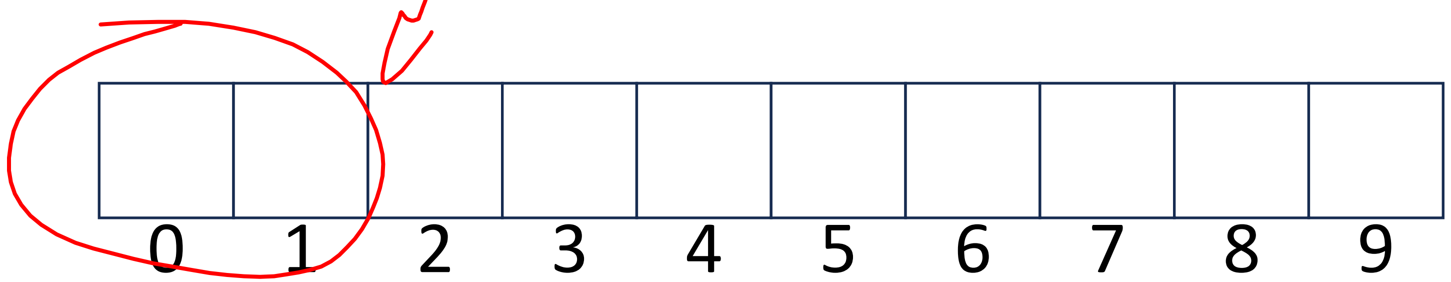
- Definition: A hash function maps objects to integers

- Should be very efficient
  - Calculating the hash should be negligible
- Should randomly scatter objects
  - Objects that are similar to each other should be likely to end up far away
- Should use the entire table
  - There should not be any indices in the table that nothing can hash to
  - Picking a table size that is prime helps with this
- Should use things needed to "identify" the object
  - Use only fields you would check for a .equals method be included in calculating the hash
  - More fields typically leads to fewer collisions, but less efficient calculation

A Bad Hash (and phone number trivia)

- $h(phone)$ = the first digit of the phone number
  - No US phone numbers start with 1 or 0
  - If we're sampling from this class, 2 is by far the most likely

206



Compare These Hash Functions (for strings)

- Let $s = s_0 s_1 s_2 \ldots s_{m-1}$ be a string of length m
  - Let $a(s_i)$ be the ascii encoding of the character s_i
- $h_1(s) = a(s_0)$ % size
- $h_2(s) = \left(\sum_{i=0}^{m-1} a(s_i)\right)$ % size
- $h_3(s) = \left(\sum_{i=0}^{m-1} a(s_i) \cdot 37^i\right)$ % size

0 – 255

spot

pots

stop

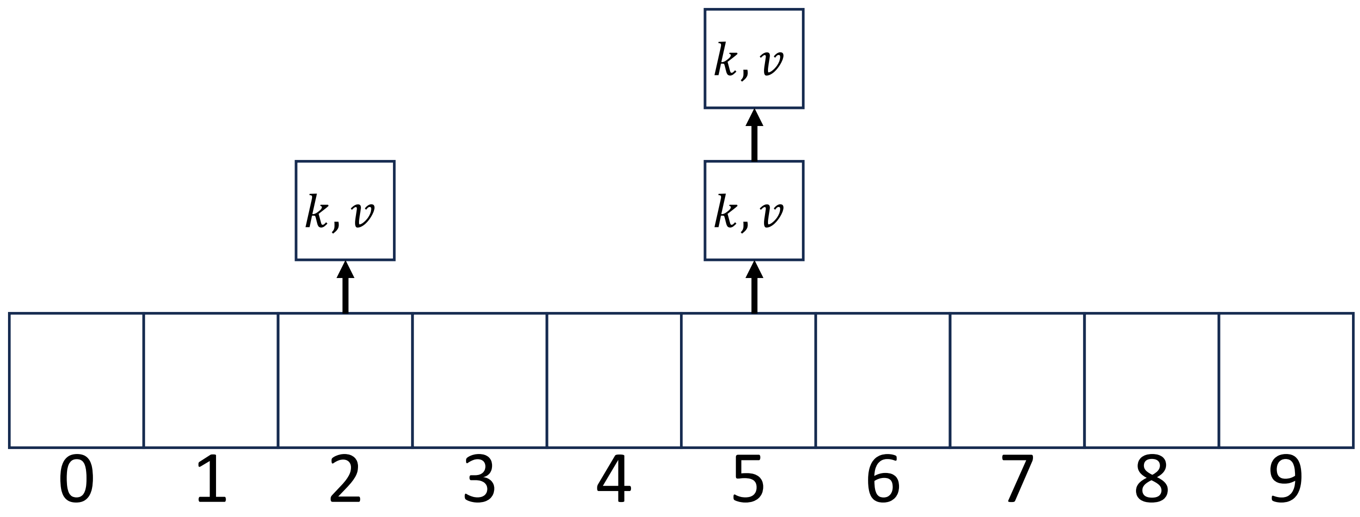
Collision Resolution

- A Collision occurs when we want to insert something into an already-occupied position in the hash table
- 2 main strategies:
  - Separate Chaining
    - Use a secondary data structure to contain the items
      - E.g. each index in the hash table is itself a linked list
  - Open Addressing
    - Use a different spot in the table instead
      - Linear Probing
      - Quadratic Probing
      - Double Hashing

0	1	2	3	4	5	6	7	8	9

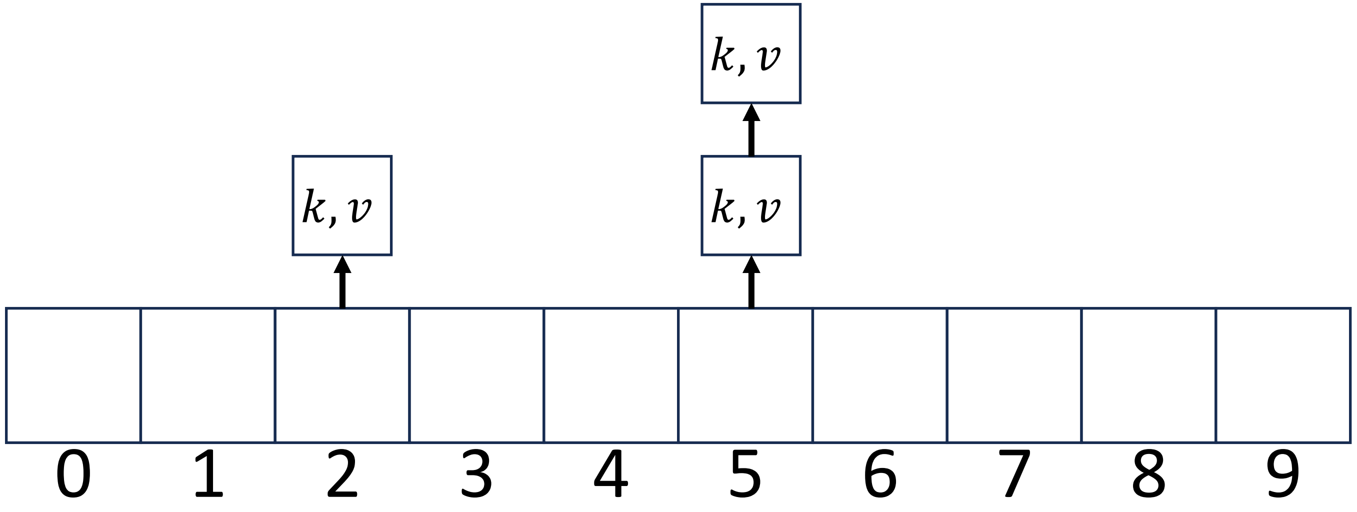
Separate Chaining Insert

- To insert k, v:
  - Compute the index using $i = h(k)\ \%\ size$
  - Add the key-value pair to the data structure at $table[i]$



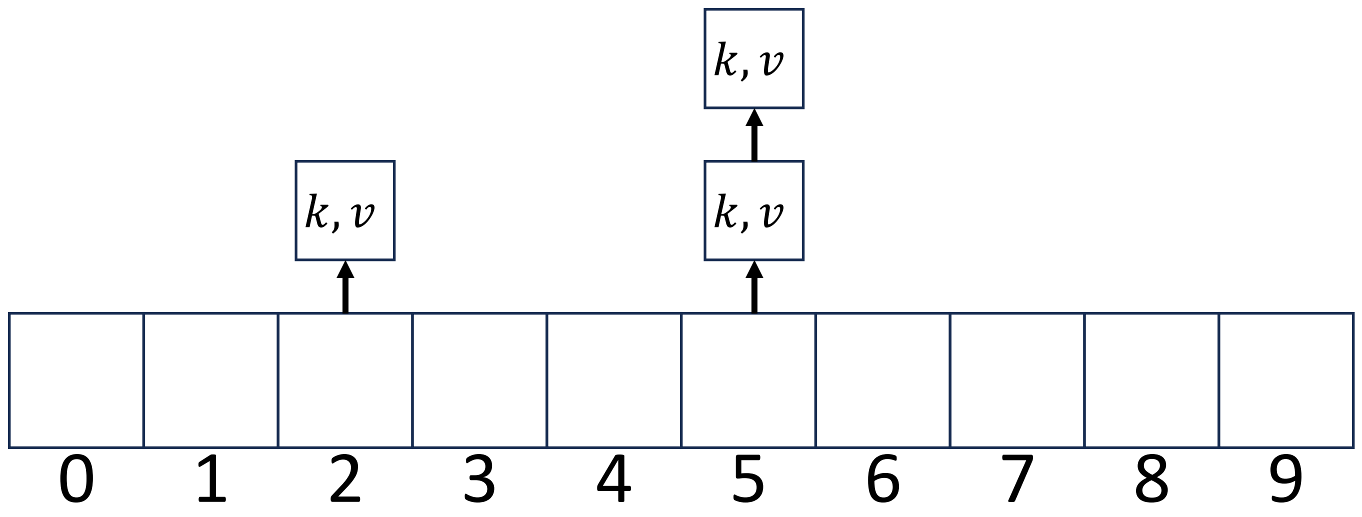
Separate Chaining Find

- To find k:
  - Compute the index using $i = h(k)\ \%\ size$
  - Call find with the key on the data structure at $table[i]$



Separate Chaining Delete

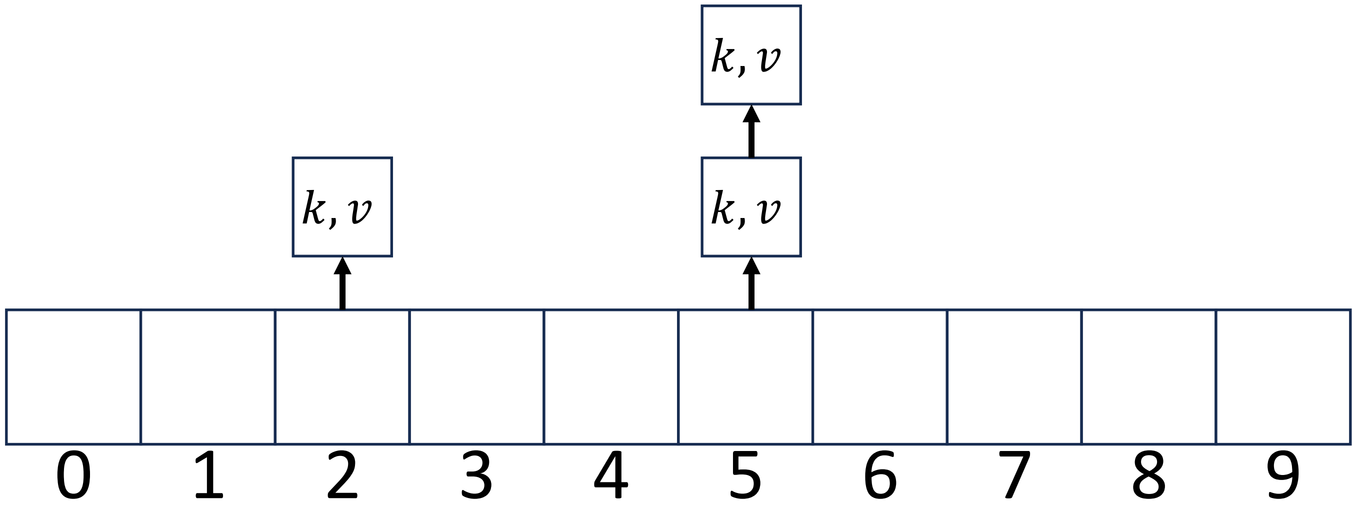
- To delete k:
  - Compute the index using $i = h(k)\ \%\ size$
  - Call delete with the key on the data structure at $table[i]$



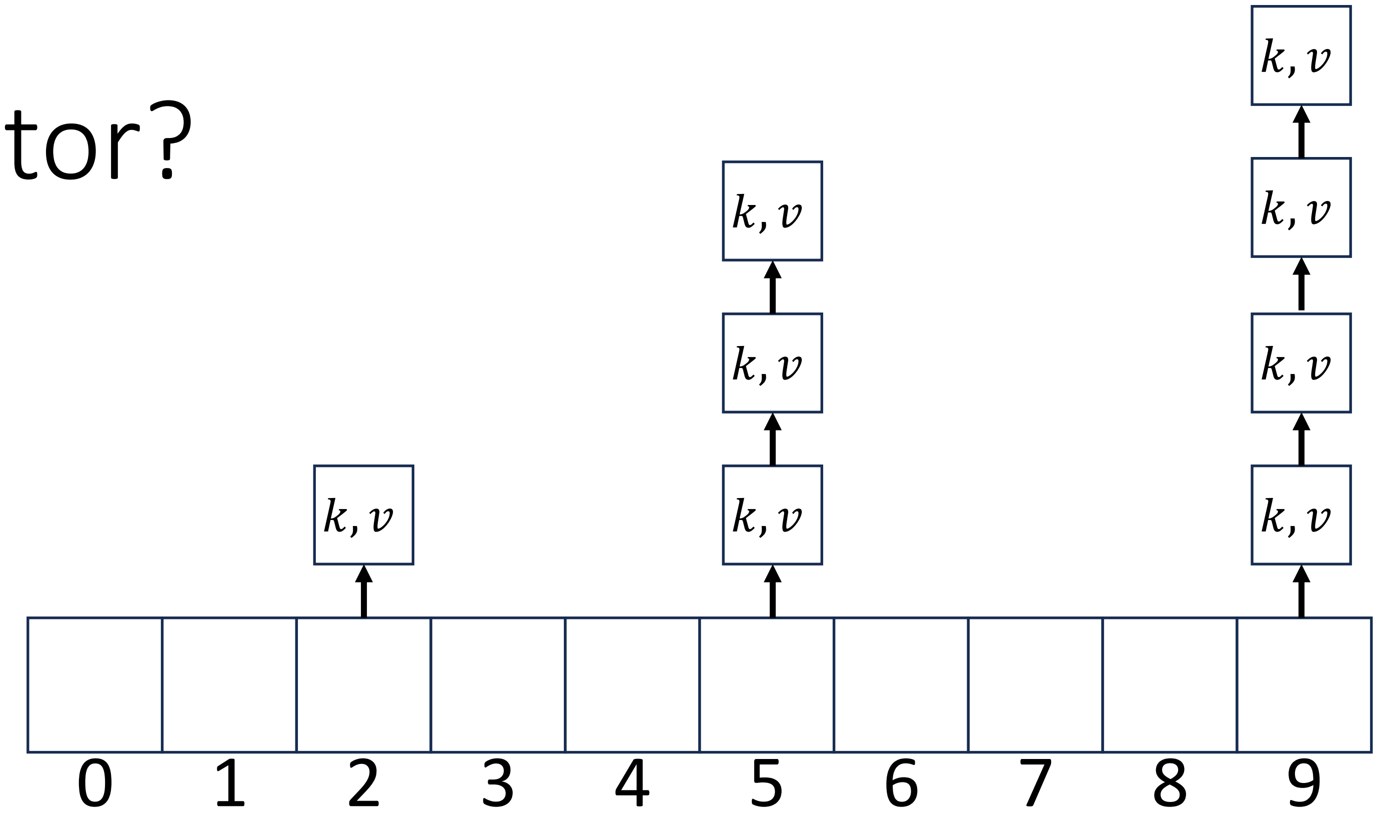
Formal Running Time Analysis

- The **load factor** of a hash table represents the average number of items per “bucket”
  - $\lambda = \frac{n}{size}$
- Assume we have a has table that uses a linked-list for separate chaining
  - What is the expected number of comparisons needed in an unsuccessful find?
  - What is the expected number of comparisons needed in a successful find?
- How can we make the expected running time $\Theta(1)$?

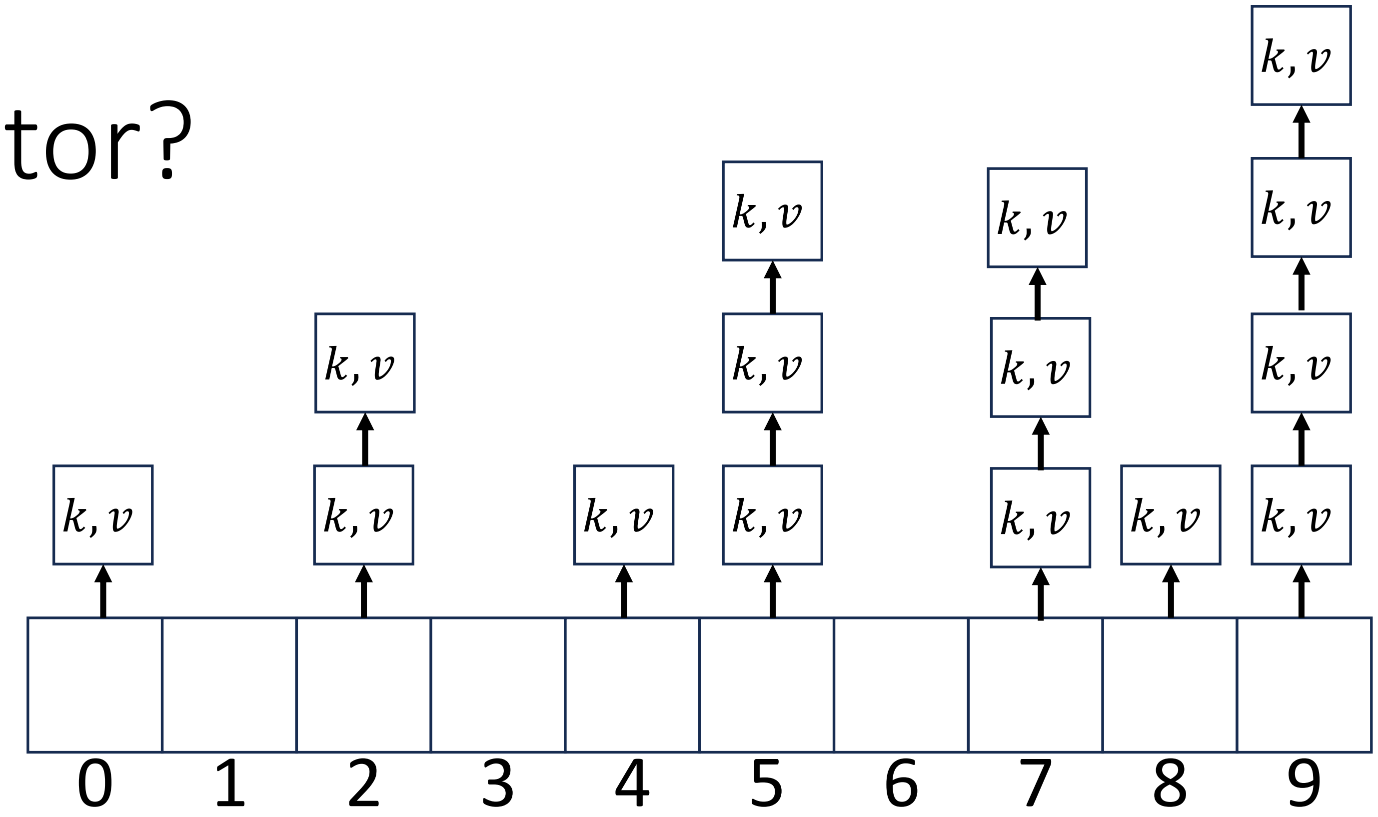
Load Factor?



Load Factor?

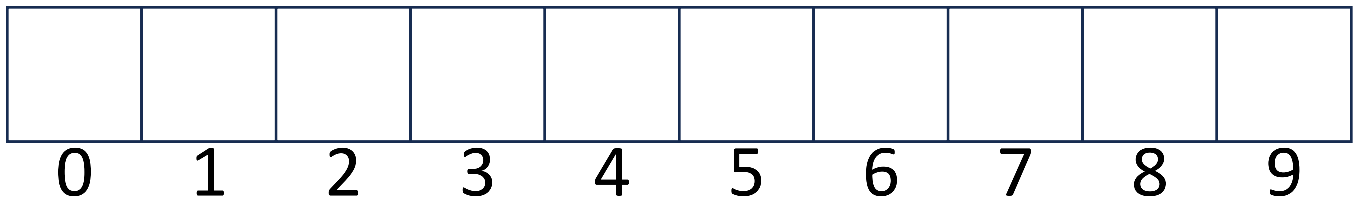


Load Factor?



Collision Resolution: Linear Probing

- When there’s a collision, use the next open space in the table



Linear Probing: Insert Procedure

- To insert k, v
  - Calculate $i = h(k)\ \%\ size$
  - If $table[i]$ is occupied then try $(i + 1)\%\ size$
  - If that is occupied try $(i + 2)\%\ size$
  - If that is occupied try $(i + 3)\%\ size$
  - …

0	1	2	3	4	5	6	7	8	9

Linear Probing: Find

- Let’s do this together!

Linear Probing: Find

- To find key k
  - Calculate $i = h(k)\ \%\ size$
  - If $table[i]$ is occupied and does not contain k then look at $(i + 1)\ \%\ size$
  - If that is occupied and does not contain k then look at $(i + 2)\ \%\ size$
  - If that is occupied and does not contain k then look at $(i + 3)\ \%\ size$
  - Repeat until you either find k or else you reach an empty cell in the table

Linear Probing: Delete

- Let’s do this together!

Linear Probing: Delete

- Let’s do this together!