CSE 332 Winter 2024 Lecture 12: Hashing

Nathan Brunelle

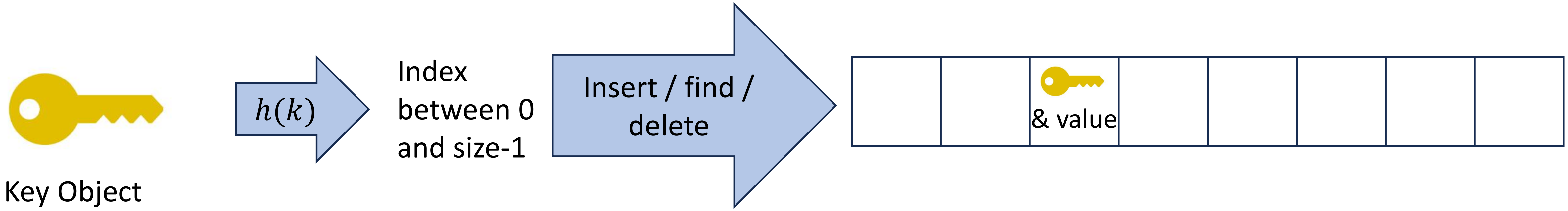
http://www.cs.uw.edu/332

Dictionary Data Structures

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)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

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



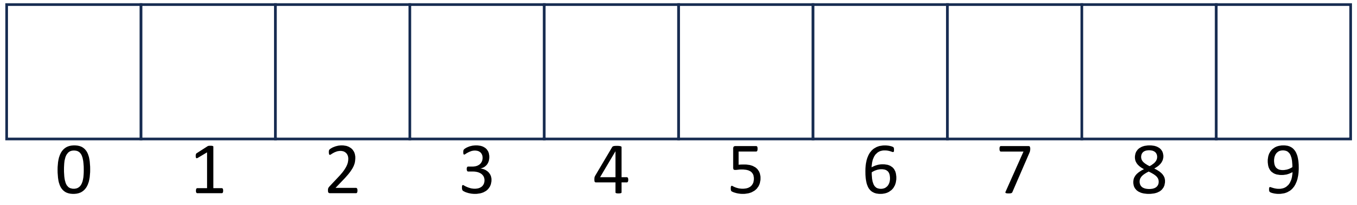
Properties of a “Good” Hash

- 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

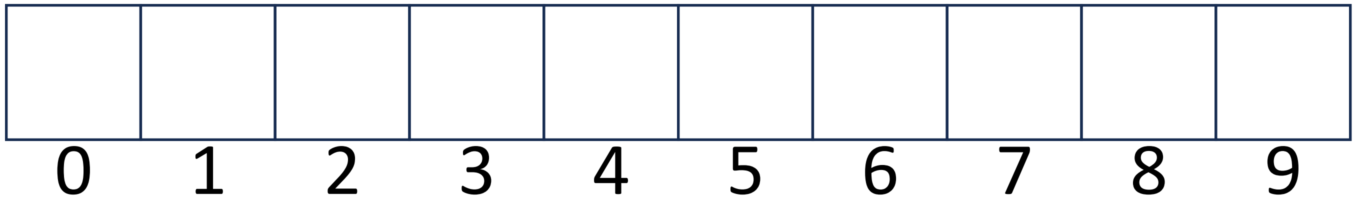


Compare These Hash Functions (for strings)

- 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
- $h_1(s) = a(s_0)$
- $h_2(s) = \left(\sum_{i=0}^{m-1} a(s_i)\right)$
- $h_3(s) = \left(\sum_{i=0}^{m-1} a(s_i) \cdot 37^i\right)$

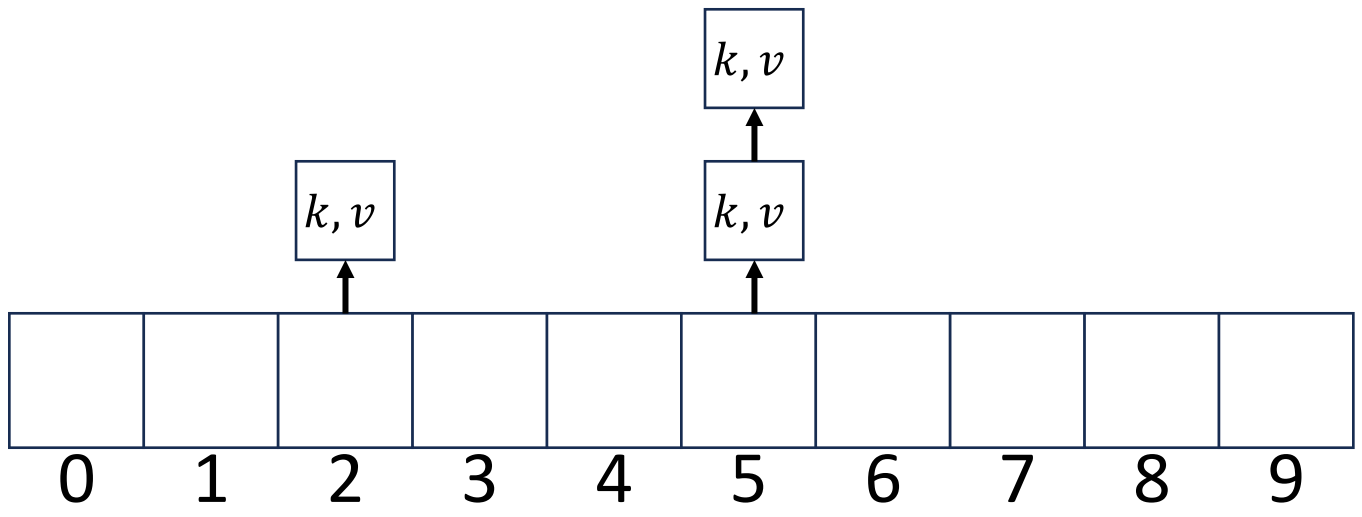
Collision Resolution

- 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



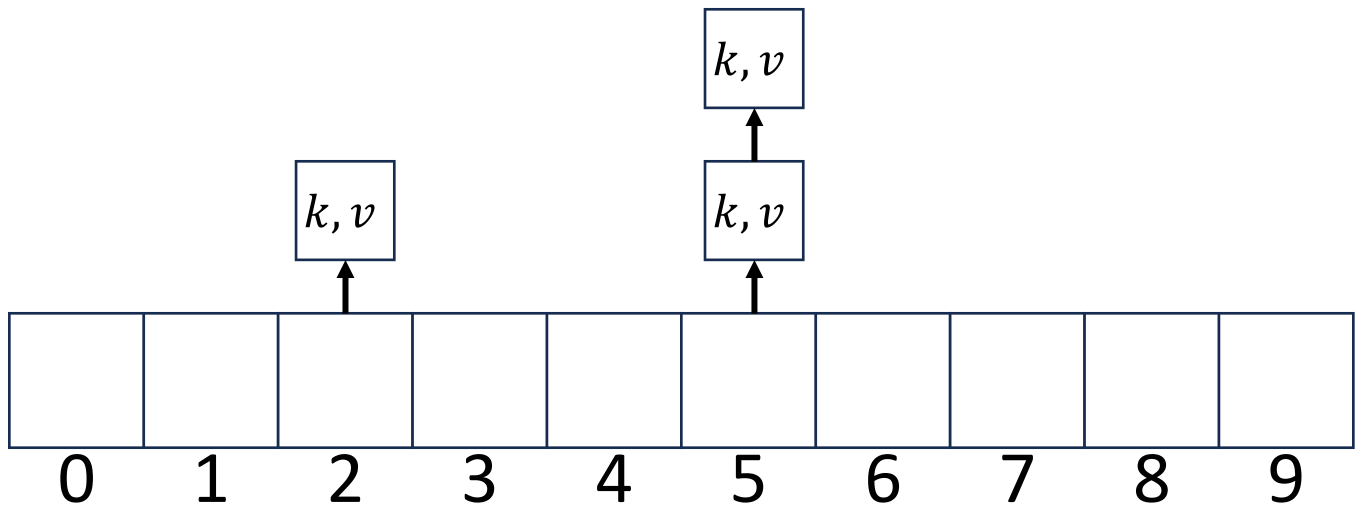
Separate Chaining Insert

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



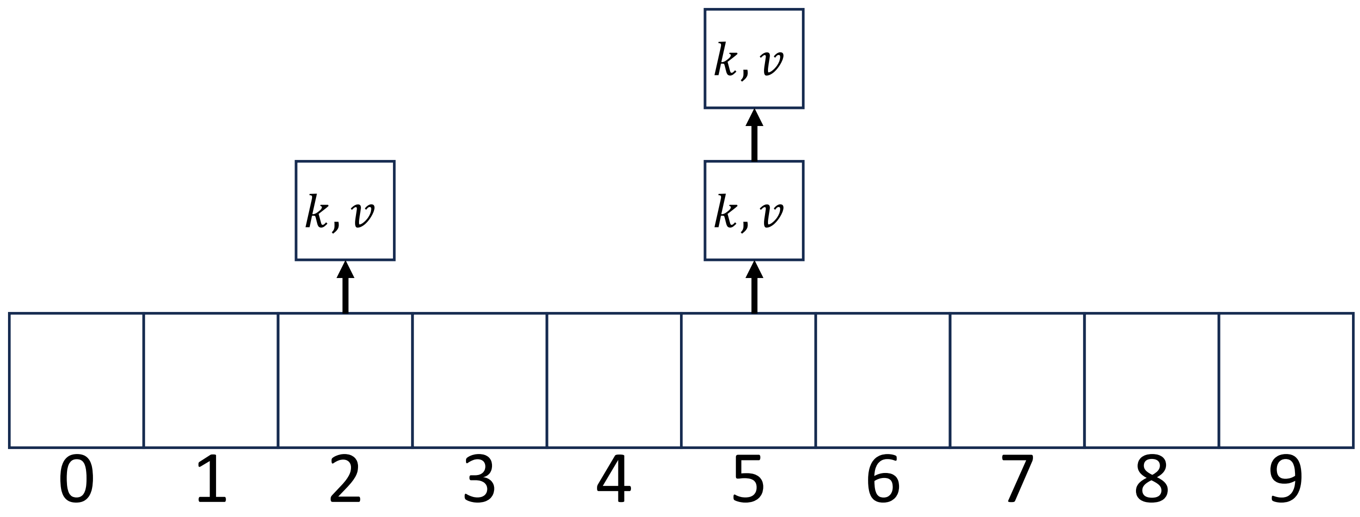
Separate Chaining Find

- 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

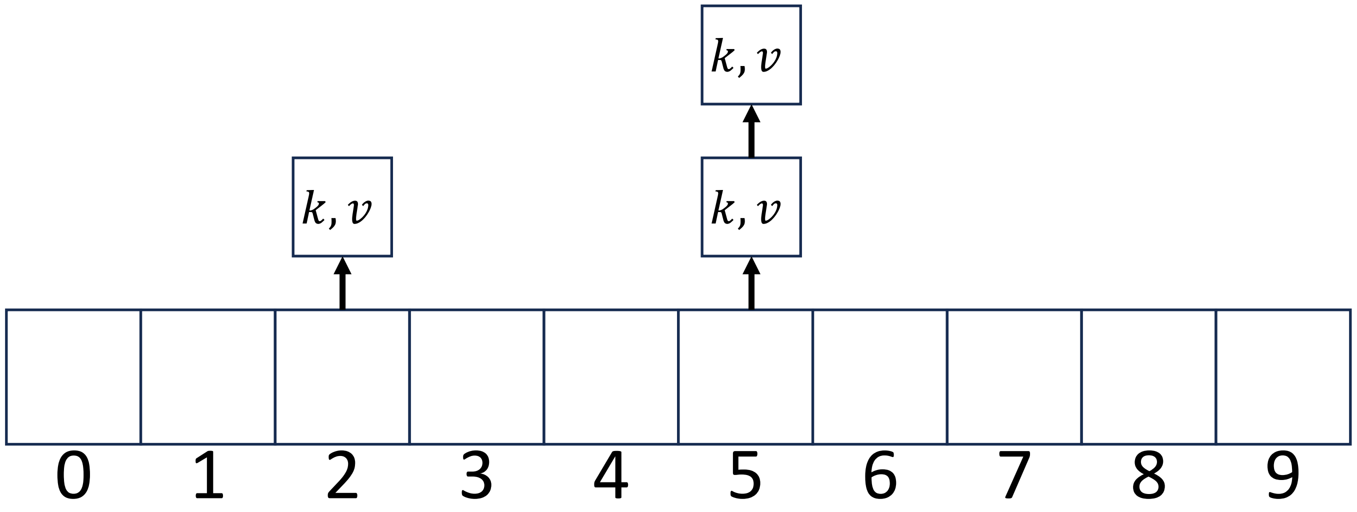
- To delete k:
  - Compute the index using $i = h(k) \ \% \ size$
  - Call delete with the key on the data structure at $table[i]$



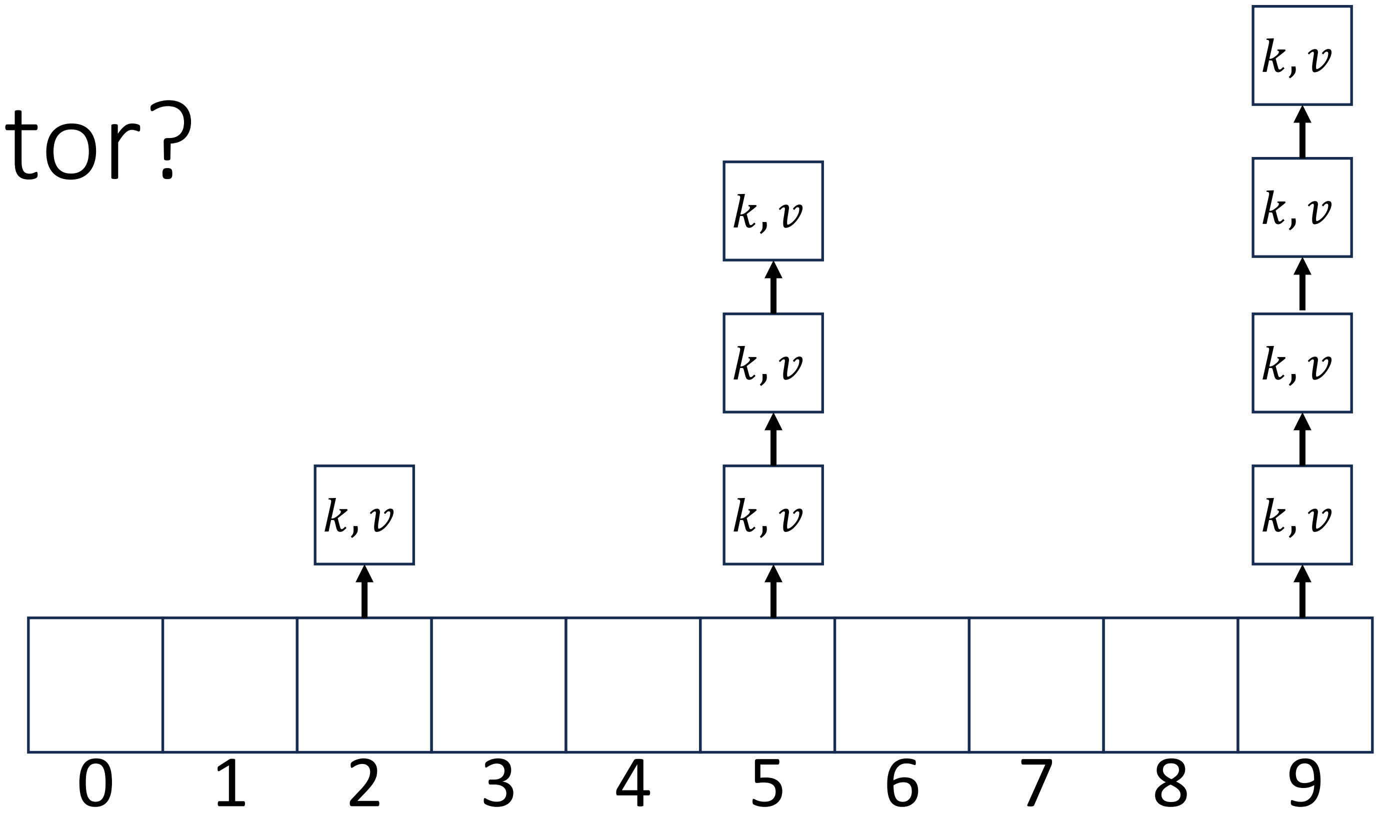
Formal Running Time Analysis

- 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 $\Theta(1)$?

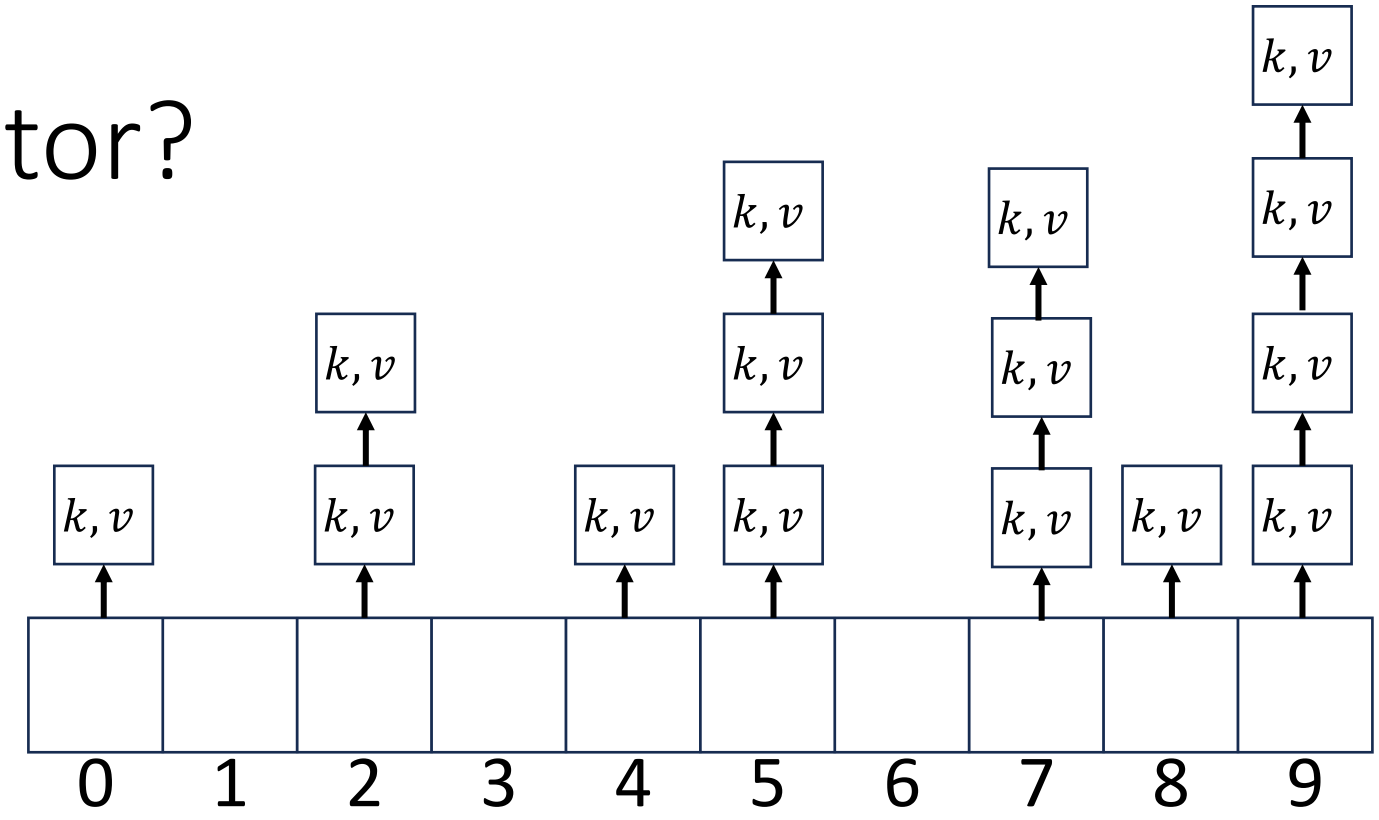
Load Factor?



Load Factor?

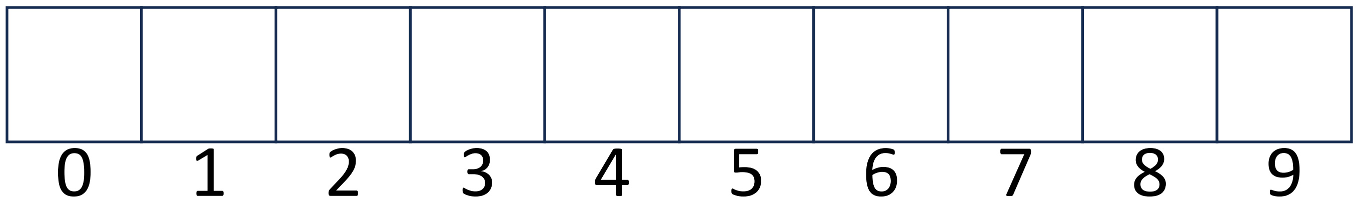


Load Factor?



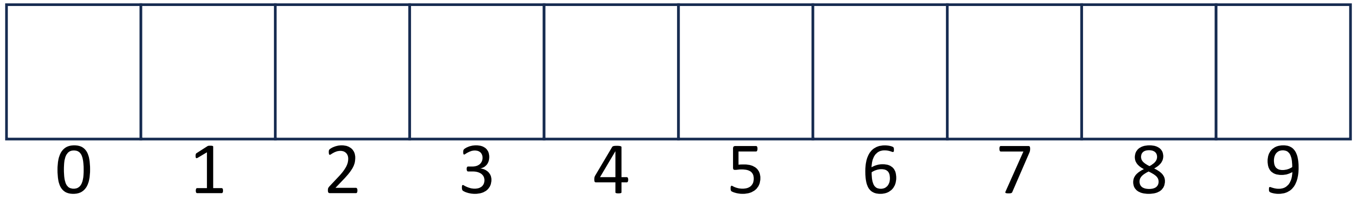
Collision Resolution: Linear Probing

- 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 $(i+3)\%\ size$
  - …



Linear Probing: Find

- Let’s do this together!

Linear Probing: Find

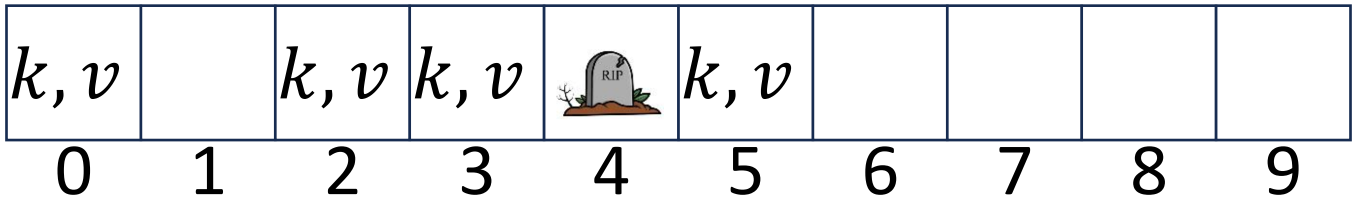
- To find key k
  - Calculate $i = h(k)\ \%\ size$
  - If $table[i]$ is occupied and does not contain k then look at $(i + 1)\ \%\ size$
  - If that is occupied and does not contain k then look at $(i + 2)\ \%\ size$
  - If that is occupied and does not contain k then look at $(i + 3)\ \%\ size$
  - 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

- Option 1: Find the last thing with a matching hash, move that into the spot you deleted from
- Option 2: Called “tombstone” deletion. Leave a special object that indicates an object was deleted from there
  - The tombstone does not act as an open space when finding (so keep looking after its reached)
  - When inserting you can replace a tombstone with a new item

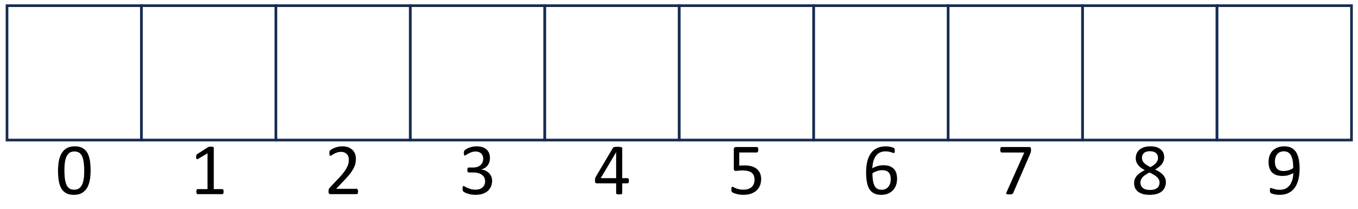


Downsides of Linear Probing

- What happens when λ approaches 1?
- What happens when λ exceeds 1?

Quadratic Probing: Insert Procedure

- To insert k, v
  - Calculate $i = h(k)\ \%\ size$
  - If $table[i]$ is occupied then try $(i + 1^2)\%\ size$
  - If that is occupied try $(i + 2^2)\%\ size$
  - If that is occupied try $(i + 3^2)\%\ size$
  - If that is occupied try $(i + 4^2)\%\ size$
  - …



Quadratic Probing: Example

- Insert:
  - 76
  - 40
  - 48
  - 5
  - 55
  - 47

0	1	2	3	4	5	6

Using Quadratic Probing

- If you probe $tablesize$ times, you start repeating the same indices
- If $tablesize$ is prime and $\lambda < \frac{1}{2}$ then you're guaranteed to find an open spot in at most $tablesize/2$ probes

- Helps with the clustering problem of linear probing, but does not help if many things hash to the same value

Double Hashing: Insert Procedure

- Given h and g are both good hash functions
- To insert k, v
  - Calculate $i = h(k)\ \%\ size$
  - If $table[i]$ is occupied then try $\left(i + g(k)\right)\ \%\ size$
  - If that is occupied try $\left(i + 2 \cdot g(k)\right)\%\ size$
  - If that is occupied try $\left(i + 3 \cdot g(k)\right)\%\ size$
  - If that is occupied try $\left(i + 4 \cdot g(k)\right)\%\ size$
  - …

0	1	2	3	4	5	6	7	8	9

Rehashing

- If your load factor λ gets too large, copy everything over to a larger hash table
  - To do this: make a new array with a new hash function
  - Re-insert all items into the new hash table with the new hash function
  - New hash table should be "roughly" double the size (but probably still want it to be prime)